HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

MASTERARBEIT

B. Sc.
Wolfgang Gehrhardt

Design Konzept für Embedded RESTful Web Services für mobile Endgeräte

2012

Fakultät **MNI**

MASTERARBEIT

Design Konzept für Embedded RESTful Web Services für mobile Endgeräte

Autor:
Wolfgang Gehrhardt

Studiengang:
Informatik

Seminargruppe:
IF09w1-M

Erstprüfer:
Prof. Dr.-Ing. Geißler

Zweitprüfer:
Prof. Dr.-Ing. Schubert

Mittweida, September 2012

Bibliografische Angaben

Gehrhardt, Wolfgang: Design Konzept fürEmbedded RESTful Web Servicesfür mobile Endgeräte, 77 Seiten, 8 Abbildungen, Hochschule Mittweida (FH), Fakultät MNI

Masterarbeit, 2012

Referat

Embedded–Systeme sind aus dem alltäglichen Leben nicht mehr wegzudenken. Mobile Endgeräte wie Smartphones oder Tablet PCs sind ein fester Bestandteil in der Landschaft des World Wide Web. Embedded Systeme in der Rolle des Web Services–Providers führen bisher noch ein Nischendasein in der IT–Landschaft. Diese Arbeit untersucht, inwieweit RESTful Web Services auf Embedded– Systemen realisierbar sind. Weiterhin wird eine einfache Lösung für die Integration in professionellen Embedded–Anwendungsfeldern in Form einer Bibliothek entwickelt und implementiert.

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

IV. Abkürzungsverzeichnis

AJAX ..... Asynchronous JavaScript and XML, Seite 68

API ..... application programming interface, Seite 68

ASCII ..... American Standard Code for Information Interchange, Seite 68

CAN ..... Controller Area Network, Seite 68

CGI ..... Common Gatway Interface, Seite 68

CSS ..... Cascading Style Sheets, Seite 68

CSV ..... comma-separated values, Seite 68

DNS ..... Domain Name System, Seite 68

FCGI ..... FastCGI, Seite 68

FTP ..... File Transfer Protocol, Seite 68

GPS ..... Global Positioning System, Seite 68

HTML ..... Hyper Text Markup Language, Seite 68

HTTP ..... Hyper Transfer Protocol, Seite 68

IANA ..... Internet Assigned Numbers Authority, Seite 68

IP ..... Internet Protocol, Seite 68

IRI ..... Internationalized Resource Identifier, Seite 68

ISP ..... Internet Service Provider, Seite 68

JMS ..... Java Message Service, Seite 68

JSON ..... Java Script Object Notation, Seite 68

MIME ..... Multipurpose Internet Mail Extensions, Seite 68

MMS ..... Multimedia Messaging Service, Seite 68

NFC ..... Near Field Communication, Seite 68

NTP ..... Network Time Protocol, Seite 68

NutOS ..... NutOS, Seite 68

OSI ..... Open System Interconnection, Seite 68

PC ..... Personal Computer, Seite 68

POSIX ..... Portable Operating System Interface, Seite 68

RAM ..... Random Access Memory, Seite 68

REST ..... Representational State Transfer, Seite 68

RFC ..... Request For Comments, Seite 68

RMI ..... Remote Method Invocation, Seite 68

ROA ..... Ressource Oriented Architecture, Seite 68

RPC ..... Remote Procedure Call, Seite 68
SMS ..... Short Message Service, Seite 68
SOA ..... Service Oriented Architecture, Seite 68
SOAP ..... Simple Object Access Protocol, Seite 68
SSH ..... Secure Shell, Seite 68
SSL ..... Secure Sockets Layer, Seite 68
TCP ..... Transmission Control Protocol, Seite 68
UDP ..... User Datagram Protocol, Seite 68
URI ..... Uniform Resource Identifier, Seite 68
URL ..... Uniform Resource Locator, Seite 68
URN ..... Uniform Resource Name, Seite 68
W3C ..... World Wide Web Consortium, Seite 68
WADL ..... Web Application Description Language, Seite 68
WAP ..... Wireless Application Protocol, Seite 68
WSDL ..... Web Services Description Language, Seite 68
WWW ..... World Wide Web, Seite 68
XML ..... Extensible Markup Language, Seite 68

1 Einleitung

1.1 Motivation

Haben PC und Spielkonsole bis vor wenigen Jahren noch den Hauptanteil der Zugangsplattformen für das Internet gebildet, hat sich heute das Feld durch Entwicklung von Smartphones und Tablet PCs stark in Richtung dieses mobilen Bereiches verschoben. Die Miniaturisierung und die Verfügbarkeit hoher Rechenleistung bei geringem Stromverbrauch erschließen in Verbindung mit mobilen Breitbandanschlüssen nicht nur im Consumer–Bereich neue Anwendungsbereiche. Der Trend im Consumer–Bereich geht dahin, mit Hilfe von Techniken, wie zum Beispiel Geo–Positioning (GPS), Near Field Communication (NFC) oder QR-Codes, eine interaktive Brücke zwischen dem Smartphone des Nutzers und seiner Umwelt zu schlagen. Gleichzeitig bieten diese Techniken auch den professionellen Einsatzbereichen von Embedded–Systemen ein großes Potential an Anwendungsmöglichkeiten. So könnte sich das Bild des Wartungstechnikers grundlegend ändern. Heute noch mit seinem speziellen Terminal–Computer oder Laptop und ebenso spezieller Software ausgerüstet, hält er zukünftig nur sein Smartphone kurz an das zu wartende System, um anschließend direkt darüber seine Wartungsarbeiten vorzunehmen. Ein noch effizienterer Weg wäre die Wartung eines Systems direkt über das Smartphone aus der Ferne über eine Web App. Der Einsatz eines Webinterface als Konfigurationsschnittstelle für Embedded–Systeme ohne ein physisches User Interface in Form einer Anzeige und Eingabegerätes, ist nicht neu. Diese Form des Nutzer–Schnittstelle ist beispielsweise in Consumer–Produkten wie Router, Drucker, Multimediageräten (z.B. Settop–Boxen) und in professionellen Embedded-Anwendungen zu finden. Dennoch spielen Web Services in eingebetteten Anwendungen eine untergeordnete Rolle, während eingebettete Systeme wie *Smartphone* und *Tablet PC* bereits einen bedeutenden Anteil der Teilnehmer des Web darstellen. Web Services im Bereich der Embedded–Systeme verfügen über sehr viel weniger Systemressourcen als Webdienste aus dem Enterprise–Umfeld. Jedoch haben sich in den Jahren gerade hier Software–Architekturen und –Techniken entwickelt und etabliert, welche heute Anwendung im Bereich des mobilen Internets finden. Häufig nutzt der Web Service eines Embedded–Systems nur rudimentär die Möglichkeiten des HTTP und damit verbunden die des *World Wide Web*. Verantwortlich dafür sind folgende Punkte:

- beschränkte Systemressourcen
- der größere Entwicklungsaufwand für eine komplett HTTP–konforme Implementierung,
- die geringe Verfügbarkeit geeigneter Middelware für Embedded–Systeme,
- die häufig redundante Entwicklung von Programmteilen in verschieden Projekten,
- der hoher Einarbeitungsaufwand für einen Embedded–Entwickler in die Web App

Entwicklung.

Die Software-Architektur und Entwicklung von Web Services ist ein viel diskutiertes und dokumentiertes Gebiet. Jedoch fehlen tiefere Betrachtungen für die Realisierung auf eingebetteten Systemen.

1.2 Arbeitsumgebung

Die vorliegende Masterarbeit entsteht als Technologie–Konzept in der Firma Taskit. Sie legt ihren Fokus auf die Entwicklung von kundenspezifischen Hardware–Lösungen für Embedded–Applikationen, die Unterstützung bei der Softwareentwicklung mit grundlegenden Softwarestacks sowie kompletten Software–Lösungen. Um für die Projekte ihrer Kunden die Möglichkeiten des mobilen Web zu erschließen, wurden verschiedene Web Services mit dazugehörigen Smartphone–Applikationen getestet und entwickelt. So beschäftigte sich zum Beispiel eine vorhergehende Diplomarbeit von Jonas Hoge in der Firma Taskit mit der Erstellung eines RESTfull Web Services für die Kommunikation über Modbus auf einem SAMDIP–7X Mikrokontroller–Modul mit 256KB Flash und 128KB RAM. Es entstand der Wunsch nach einer einheitlichen, wiederverwendbaren Lösung für Embedded Web Services. Dafür standen drei firmeneigene Hardware–Module zur Verfügung.

- PicosG20
- NanosG20
- SAMDIP-7X

1.3 Ziel

Die vorliegende Arbeit soll aufzeigen, wie sich Embedded–Applikationen in das moderne *World Wide Web* (WWW) einbinden lassen. Der Fokus liegt auf der Integration mittels RESTful Web Services und welche Möglichkeiten sich durch Verwendung neuer HTML5–Technologien ergeben. In diesem Zusammenhang soll untersucht werden, ob und inwieweit sich der etablierte Software Architekturstil REST, aus dem Enterprise Umfeld, auf Embedded–Systeme mit begrenzten Hardware–Resscourcen übertragen lässt. Dabei werden die, in der Literatur häufig abstrakten, Betrachtungen des RESTful Software Designs auf konkrete, problemspezifische Implementierungen für Embedded–Systeme hin untersucht. Wenn erforderlich, sollen Designrichtlinien für RESTful Web Services auf Basis von Embedded–Systemen definiert werden. Auch werden damit einhergehende mögliche Szenarien der Netzwerkinfrastruktur zur Verteilung der Aufgaben bzw. Web Services auf verschiedene Embedded–Systeme angesprochen. Als weiteres Ziel der Einführung einer RESTful Software–Architektur sollen zeitaufwändige, ständig wiederkehrende bzw. sich wiederholende Implementationen vermieden werden.

Über einheitliche Softwareschnittstellen soll die Einarbeitung in bestehende Web Service Projekte zum Beheben von Störungen oder für Erweiterungen und Anpassungen optimiert werden. Als praktischer Anteil soll eine Sammlung von Softwaremodulen in Form einer C-Bibliothek die REST-konforme Entwicklung von Embedded Web Services erleichtern.

1.4 Aufbau der Arbeit

Das zweite Kapitel erläutert zum einen kurz die grundlegenden Techniken dieser Arbeit. Zum anderen soll eine Einführung in den Bereich der Embedded–Systeme gegeben und das Verständnis für spätere Vergleiche von herkömmlichen Web Services und jenen speziell für eingebettete Systeme gefördert werden.

Nach Erläuterung der Grundlagen, wird im 3. Kapitel auf Web Service Architekturen näher eingegangen. Hierbei werden zunächst die möglichen Anwendungsfälle und Anforderungsprofile erörtert. Daraufhin folgt die Untersuchung möglicher Architekturstile für Web Services allgemein. Dabei werden die beiden großen Vertreter der Web Service Architekturen SOA und ROA erklärt und deren Verwendung und Nutzbarkeit hinsichtlich der begrenzten Hardware Ressourcen von Embedded–Plattformen herausgestellt. Abschließend zum Kapitel *Web Service Architekturen* führt der letzte Abschnitt mögliche Architekturen an, mit denen RESTful Web Services umgesetzt werden können. Es werden dabei mögliche Kombinationen aus verschiedenen Software-, Hardware- und Netzwerkkonfigurationen gezeigt.

Das 4. Kapitel widmet sich der Repräsentation von Daten in einer Web Service Architektur. Die Berücksichtigung der Anforderungen an Datenrepräsentationen in Verbindung mit Embedded-Systemen bildet hier das zentrale Thema. Es werden zu Beginn die verschiedenen Ebenen der Datenrepräsentation in einem webbasierenden System definiert. Im Fokus der Betrachtung liegt die Wahl geeigneter Datenrepräsentationsformate für die Kommunikation zwischen Embedded Web Services und mobilen Endgeräten. Abschließend werden einige universal einsetzbare Formate vorgestellt.

Anschließend im 5. Kapitel 5 werden zunächst die Grundzüge des RESTful Designs bzw. das Wesen einer *Resource Orientated Architecture* festgehalten. Danach rückt die Identifikation und das Design von Ressourcen für einen Web Service in den Mittelpunkt. Hier soll aufbauend zum vorhergehenden Kapitel 4 die Verbindung von Ressourcen zu deren Repräsentation erfolgen. Dabei liegt auch hier der Schwerpunkt auf Design–Ansätzen speziell für das *Mobile Client & Embedded Web Service* Szenario. Der letzte Teil des Kapitels thematisiert Funktionalitäten des HTTP, welche für die Planung und Umsetzung von Embedded Web Services entscheidend sein können.

Mit dem 6. Kapitel wird das Gegenstück zum RESTful Embedded Web Service, der HTTP–Client, und seine Implementierung betrachtet. Konkret liegt der Blickpunkt auf den mobilen Clients, unabhängig von ihrer Form (z.B. Smartphones, Tablet PCs, ...). In diesem Zusammenhang werden mögliche Client– Applikations–Arten diskutiert. Abschließend wird die konkrete Implementierung einer Web–App als HTTP–Client mittels jQuery-Mobile und HTML5 erläutert.

Kapitel 7 dokumentiert die, parallel zu dieser Arbeit entwickelte, *RESTful Web Service Libraray* (librws). Es werden hier sowohl das Konzept, Design als auch die Implementierung dargelegt.

Das letzte Kapitel widmet sich der Analyse und Fehlersuche der in dieser Arbeit beschriebenen RESTful Web Service Architektur. Es werden eingangs einige Voraussetzungen und Vorüberlegungen für das Testen und Debuggen dargelegt. Daran anschließend wird das Testen, besonders das automatisierte Testen, angesprochen und die für diese Arbeit verwendeten Techniken und Werkzeuge vorgestellt. Auf das Feststellen von Fehlern mittels dieser Vorgehensweise, folgt das Auffinden und Beseitigen von Fehlern im Programm. Dazu stellt der letzte Abschnitt zunächst typische Fehler und Fehlerquellen vor. Für das Auffinden und Beheben dieser Fehler werden bekannte und für diese Arbeit benutzte Debugg–Werkzeuge sowie deren Verwendung vorgestellt.

2 Grundlagen

Zum Verständnis dieser Arbeit ist es nötig, vorab auf drei elementare Themengebiete einzugehen. Das sind:

- Das World Wide Web
- Das mobile Web
- Embedded–Systeme

Im Folgenden werden Grundlagen und Begrifflichkeiten dieser Bereiche erläutert.

2.1 Das World Wide Web

Im Jahr 1989 wurde das zukünftige Worl Wide Web (WWW) erstmals von Tim Berners-Lee im Text [Berners-Lee(1990)] beschrieben. Daraus entstanden die ersten Versionen der grundlegenden Techniken des heutigen WWW. 5 Jahre später, 1994, wurde das World Wide Web Consortium (W3C) von ihm gegründet. Das W3C versteht sich bis heute als Komitee zur Entwicklung und Standardisierung von Web–Technologien. Es besteht aus ca. 300 verschiedenen Mitglieds-Organisationen (z. B. Firmen, Universitäten), die die Arbeit des Consortiums unterstützen. Sie ermitteln die Bedürfnisse, Notwendigkeiten und Möglichkeiten der Internet–Benutzer und führen diese zusammen. Gemeinsames Ziel der Beteiligten ist, einheitliche Standards für die Nutzung des Internets einzuführen und somit zu erreichen, das Potential, welches es bietet, optimal zu nutzen.

2.1.1 Struktur des Internets

Das Internet steht für das große globale Computernetzwerk, bestehend aus vielen einzelnen öffentlichen und privaten Netzwerken. Es stellt als Netzwerkinfrastruktur das Rückgrat des WWW dar. Jeder netzwerkfähige Computer kann sich über das *Internet Protocol* (IP) in diesen Verbund von Computern einklinken und Informationen über verschiedene Protokolle und Dienste austauschen. Die Infrastruktur des Internets wird durch die Netzbetreiber[1] gestellt. Der Zugang zum Internet wird ebenfalls von diesen ISPs vermittelt. Ein Teilnehmer (Client), der mit dem Internet verbunden ist, erhält dafür eine eindeutige Adresse, die IP–Adresse. Für die Kommunikation im Internet bildet das Internet Protocol die Grundlage und stellt nach dem OSI–Schichtenmodell[2] die Vermittlungsschicht (Ebene 3) dar. Darauf aufbauend bilden die Protokolle TCP und

[1] Internet Service Provider (ISP)
[2] Open System Interconnection Reference Model

UDP die Transportschicht (Ebene 4) im OSI–Schichtenmodell. Der Austausch von Anwendungsdaten wird über verschiedene Anwendungsprotokolle in den Ebenen 7-5 des OSI-Schichten–Modells realisiert. Als typische Vertreter für die Anwendungsprotokolle können folgende genannt werden:

- **D**omain **N**ame **S**ystem
- **H**yper**T**ext**T**ransfer**P**rotocol
- **F**ile**T**ransfer**P**rotocol
- **S**imple**M**essage**T**ransport**P**rotocol
- **N**etwork**T**ime**P**rotocol

.

Um einen Dienst im Internet in Anspruch nehmen zu können, muss dem Teilnehmer (Client) die IP–Adresse des Anbieters bekannt sein. Eine für den Menschen einfachere und semantisch zugänglichere Variante dieser IP–Adresse ist der *Domain Name*. Jeder, der im Internet einen Service bereitstellen möchte, kann sich einen solchen *Domain Namen* reservieren. Über das *Domain Name System* (DNS) können diese Namen zu IP-Adressen und umgekehrt aufgelöst werden.

Die Infrastruktur des globalen Netzwerks beinhaltet auch Lastverteilungs- und Optimierungsmechanismen. Die Lastverteilung wird beispielsweise über Router realisiert, die an Knotenpunkten des Internets entscheiden, welche Netzwerkroute ein Datenpaket zum Zielpunkt nimmt. Für die Entlastung bzw. zur Reduzierung redundanten Datentransfers setzen ISP Proxies ein. Diese sollen Antworten von Diensten auf gleiche Anfragen zwischenspeichern und anstelle des eigentlichen dienstanbietenden Servers antworten. Dieses sogenannte Caching-Verfahren kann so die Antwortzeit für den Client drastisch verringern und gleichzeitig den dienstanbietenden Server entlasten. In Verbindung mit dem HTTP können durch diese Eigenschaft folgende Vorteile erreicht werden:

- reduzierte Serverlast durch Minimierung der zu bearbeitenden Anfragen
- reduziertes Transfervolumen beim Dienstanbieter oder Client, je nachdem, an welcher Stelle des Kommunikationsweges das Caching einsetzt
- reduzierte Netzwerklast, da Anfragen und Antworten nicht durch das gesamte Internet geleitet werden .

Ein solcher Proxy als Caching–Lösung kann auch schon im lokalen Netzwerk vor einen HTTP–Server geschaltet werden und so hier die Webserver entlasten.

2.1.2 Resource Identifier und Locators

Eine der 3 Basis–Technologien des WWW sind die sogenannten Resource Identifier. Durch sie ist es erst möglich, Inhalte im WWW ausfindig zu machen. Sie können in verschiedene Gruppen untergliedert werden. Alle gehören aber in die Menge der Unified Resource Identifier (URI). Die beiden wichtigsten und bekanntesten Resource Identifier sind URI und URL und werden im Folgenden kurz erläutert.

URI

Die URI bildet die Obermenge aller Resource Identifier. Sie legt die Syntax Komponenten für Resource Identifier fest. Die URI besteht aus einem Schema und einem Schema–spezifischen Teil. Untergruppen können sich auf spezielle Formen der URI beschränken und so den Eindruck erwecken, dass ihnen eine andere Syntax zugrunde liegt. Dies ist jedoch nicht der Fall, einige Teile einer URI sind optional und können weggelassen werden. Die vollständige Definition der URI ist in der [Berners-Lee et al.(1998)Berners-Lee, Fielding, and Masinter]. Eine URI verwendet den US-ASCII Zeichensatz und besteht aus vier Komponenten, welche nicht von jedem URI-Typ verwendet werden müssen. Der Aufbau einer URI kann demnach wie folgt beschrieben werden: sheme : authority path ? query

scheme Das Schema (scheme) dient zur Identifizierung eines speziellen URI-Schemas und den damit verbunden Regeln für den schema–spezifischen Teil.

authority Dieser Teil beinhaltet die Adressinformation für eine Ressource wie:

- Nutzername
- Passwort
- Host
- Hostport

Die Angabe eines Passwortes ist hier zwar vorgesehen, jedoch sollte dies bei nicht sicheren Verbindungen unterlassen werden. Als nicht sicher sollen Verbindungen gelten, in denen die URI unverschlüsselt gesendet wird, wie es bei HTTP über TCP/IP der Fall ist.

path Der Pfad stellt die hierarchische Komponente zur Identifizierung von Ressourcen in der URI dar.

query Die *query* bietet die Möglichkeit eine String mit beliebigen Inhalt, ausgenommen einiger Sonderzeichen (s. [Berners-Lee et al.(1998)Berners-Lee, Fielding, and Masinter] Seite 15), an eine Ressource zu übergeben. Auch wenn es im Fall eines statischen Text–Dokuments wie einem PDF nicht besonders nützlich erscheint, so ist diese Funktion ein wichtiger und oft verwendeter Mechanismus in Web Services.

URL

Die URL ist historisch gesehen älter als die URI. Sie gehörte zu den von Berners-Lee erfundenen grundlegenden Technologien des WWW. Die URL ist eine Untergruppe der URI . Auch sie adressiert eine Ressource im Internet. Während die URI allgemein eine Ressource adressiert, liegt der entscheidende Unterschied der Uniform Resource Locator darin, dass er eine spezielle Form einer Ressource identifiziert.Vom Aufbau bedient sie sich der Syntax der URI. Die URL ist in der RFC1738 festgehalten und kann unter [Berners-Lee et al.(1994)Berners-Lee, Masinter, and McCahill] eingesehen werden.

IRI

Der *Internationalized Resource Identifier* wurde als Ergänzung zur URI geschaffen, um erweiterte Zeichensätze in einer URI zu unterstützen. So sind zum Beispiel deutsche Umlaute in einer IRI möglich. Eine IRI nutzt den ISO10646 als Zeichensatz. In der RFC3987 wird die IRI definiert und ist unter [Duerst and Suignard(2005)] zu finden.

2.1.3 Das Hypertext Transfer Protocol

Das HTTP wurde am CERN im Jahr 1989 entwickelt und stellt bis heute eine von drei grundlegenden Technologien des WWW dar. Die Abkürzung HTTP steht für *Hpyertext Transfer Protocol*. Erfunden zum Austausch von Dokumenten, hat sich das HTTP zu einer Basistechnologie des Datenaustausches in Verteilten Systemen im heutigen Internet entwickelt. Der einfache Aufbau macht es zu einer „Sprache“, die heute beinahe jedes Gerät mit Verbindung zum Internet „sprechen“ kann. Es gibt eine Vielzahl an verfügbaren Client– und Server–Implementierungen des Protokolls. Die bekannteste Gattung von HTTP–Client–Programmen stellen die Web-Browser dar. Sie bieten dem Nutzer ein generisches Interface für die Navigation und Visualisierung von Inhalten im Web. Andere Gruppen sind Kommandozeilen–Werkzeuge oder Middleware bzw. Frameworks in Form von Bibliotheken. Diese sind für fast jede Programmiersprache und Laufzeitumgebung verfügbar. Gleiches gilt für die HTTP–Server–Implementierungen. Die bekannteste Klasse ist hier die der Webserver. Zu ihnen gehören Apache, ngix, lighttpd oder IIS. Diese Tatsache macht das Protokoll heute zur führenden Informationsaustausch–Technologie.

Im OSI–Schichtenmodell ist es in der Ebene 5-7 angesiedelt und setzt auf dem TCP/IP Protokoll auf. Zur Zeit dieser Arbeit ist das Protokoll in der Version 1.1 in Anwendung und wird in der [Fielding et al.(1999)Fielding, Gettys, Mogul, Frystyk, Masinter, Leach, and Berners-Lee] beschrieben. Der TCP-Port 80 wird als Standard–Port für die Kommunikation verwendet, alternative bekannte Varianten sind Ports in der Region 8080-8089. Das HTTP besitzt die folgende Eigenschaften:

- textorientiert ASCII
- zustandslos
- synchron
- Client–Server / Request–Reply Model
- idempotenz der HTTP–Methoden GET, HEAD, OPTION

Jedes Dokument, also jede Ressource, hat einen Zustand, welcher durch einen HTTP-Code repräsentiert wird (s. Anhang A). Die Eigenschaft der Zustandslosigkeit des HTTP bezieht sich jedoch nicht auf diesen Zustand der Ressource, sondern den Anwendungszustand. Ein HTTP–Server behält sich keine Informationen über den momentanen Zustand des Clients in einer Anwendung. Der Client ist für die Verwaltung und Navigation zwischen verschiedenen Anwendungszuständen verantwortlich. Erweiterungen wie Cookies im HTTP-Protokoll weichen diese Einschränkungen zu Lasten der Skalierbarkeit einer Anwendung auf. Näher auf das Problem der Zustandslosigkeit des HTTP und zur Realisierung der Anwendungszustände wird in Verbindung mit REST–konformen Web Service (RESTful Web Service) im Kapitel 3 eingegangen. Aus dem zugrunde liegenden Client–Server Modell des HTTP ergibt sich ein synchroner Kommunikationsablauf. Das bedeutet, die Kommunikation läuft nach dem Frage–Antwort–Schema ab und ist vom Client initiiert. HTTP besitzt nach dem derzeitigen Standard 1.1 keinen Mechanismus für eine serverseitige Benachrichtigung eines Clients. Diese Variante der Kommunikation wird auch unter den Begriffen *HTTP–Push* oder *HTTP server push* geführt. Für die Realisierung eines solchen Szenarios gibt es verschiedene alternative Ansätze, welche unterschiedlich stark verbreitet sind. Die folgenden 3 Varianten sind die gegenwärtig gebräuchlichen bzw. vielversprechendsten.

- Server–Send–Events ([Hickson(2012)])
- WebSockets ([Hickson(2009)])
- Emulation über Plugins (z.B. Java, Flash)

Protokollaufbau

Die Kommunikation läuft beim HTTP über den synchronen Austausch von HTTP–Nachrichten (HTTP–Message) nach dem Client–Server Schema ab. Das bedeutet, ein HTTP–Client formuliert eine Anfrage, nachfolgend als HTTP–Request bezeichnet, an einen HTTP–Server. Der HTTP–Server wertet den HTTP–Request aus und erzeugt eine ent-

sprechende Antwort, im weiteren Verlauf HTTP–Response genannt. Ein HTTP–Request und HTTP–Response stellen demnach jeweils eine HTTP–Message dar. Eine HTTP-Message enthält einen Message–Header und –Body.

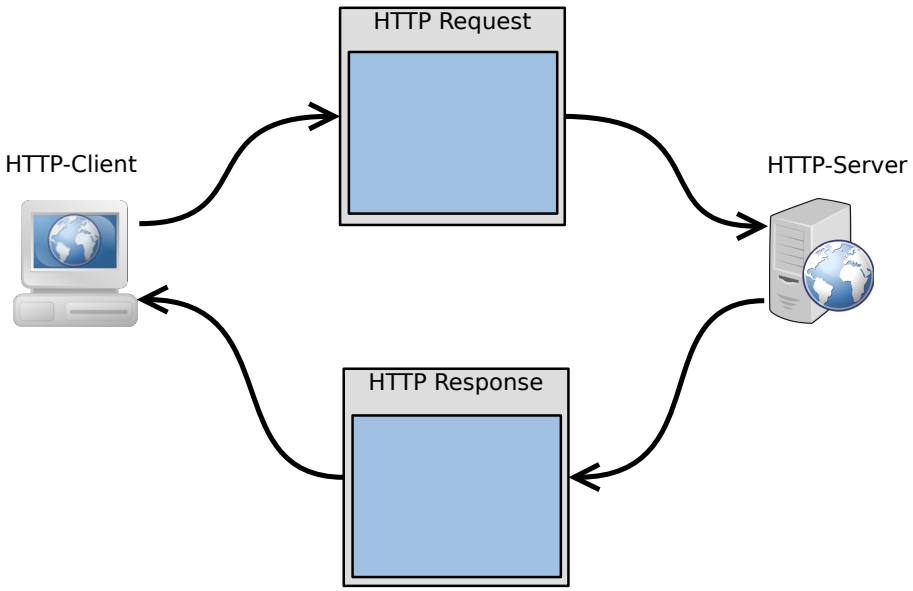

Abbildung 2.1: Ablauf der Kommunikation über das HTTP

Die Message–Header sind einfache Name–Wert Paare die mit einem „\r\n“ abgeschlossen werden. Sie können in drei Kategorien unterteilt werden:

Request–Header: sind Header, die nur in einem HTTP–Request vorkommen.

Response–Header: sind Header, die nur in einem HTTP–Repsonse vorkommen.

Entity–Header: bezeichnen Header, die sich auf den Inhalt des HTTP–Message–Body beziehen, also der HTTP–Request/Response–Entity.

Eine detaillierte Aufzählung und Beschreibung der HTTP Request– und Response–Header sind in Kapitel 14 der [Fielding et al.(1999)Fielding, Gettys, Mogul, Frystyk, Masinter, Leach, and Berners-Lee] gegeben.

Theoretisch kann jeder HTTP–Request eine HTTP–Entity enthalten, jedoch entscheidet der Server ob und wann er sie annimmt. Das Vorhandensein einer HTTP–Entity bei einem HTTP–Response hingegen hängt zum einen von der Implementierung des HTTP–Servers und zum anderen vom Response–Code ab. (s. [Fielding et al.(1999)Fielding,

Gettys, Mogul, Frystyk, Masinter, Leach, and Berners-Lee]) . Die sogenannte Entity eines HTTP–Request oder Responses können Daten in einem beliebigen MIME–Type sein. Mehr zu den MIME–Types und ihre Bedeutung wird im Kapitel 4 erläutert.

Die beiden HTTP-Messages HTTP–Request und HTTP–Response unterscheiden sich in der ersten Zeile der HTTP–Message. Die Request–Line bezeichnet die erste Zeile eines HTTP–Request. Sie setzt sich aus drei Teilen zusammen:

HTTP–Method: Gibt die Zugriffsmethode für die in der URI adressierte Ressource an. Das HTTP 1.1 definiert GET, HEAD, OPTION, PUT, POST, DELETE, TRACE und CONNECT. Näher auf die Bedeutung der einzelnen Methoden wird im Kapitel 5 zusammen mit den RESTful–Paradigmen eingegangen.

URI: Sie adressiert die Ziel–Ressource für den Request.

HTTP–Version: Sie gibt dem HTTP–Server an, welche Version des HTTP der Client verwendet.

Als *status line* wird die erste Zeile in einem HTTP–Response bezeichnet. Sie besteht aus den drei Elementen HTTP–Version des Servers, gefolgt von einem 3–stelligen Status–Code und daran anschließend eine Zeichenkette mit einer Beschreibung in Textform. Die HTTP–Status–Codes werden in fünf Kategorien unterteilt:

100er Gruppe: dient zur Aushandlung der Verbindung zwischen HTTP–Client und HTTP–Server.

200er Gruppe: teilt dem Client eine erfolgreiche Bearbeitung seines Requests mit.

300er Gruppe: teilt dem Client mit, wie er zu einer angefragten Ressource gelangt.

400er Gruppe: wird verwendet, um dem Client auf einen Fehler in seiner Anfrage hinzuweisen.

500er Gruppe: informiert den Client über einen serverseitigen Fehler während der Verarbeitung des HTTP–Requests.

Eine vollständige Auflistung der HTTP–Status–Codes ist im Anhang A zu finden.

Fähigkeiten des HTTP–Protokolls

Das HTTP verfügt über viele Mechanismen die für den Informationsaustausch benötigt werden. Zu den bekanntesten zählen:

- Autorisierung
- Content-Negotiation
- Datenkompression

- Caching

Nähere Erläuterung finden diese im Zusammenhang mit der Diskussion ihrer Bedeutung für RESTful Web Service in Embedded–Systemen.

Einige weitere Mechanismen sind nur wenig bekannt. So zum Beispiel verfügt das HTTP über Möglichkeiten, die Verbindung zwischen Client und Server über die eigenen, der Natur des Protokolls gegebenen, Grenzen hinweg zu erweitern. Diese Funktion kann mit Hilfe des Upgrade–Headers angefordert werden. Dieser Mechanismus findet Verwendung in der Umsetzung von Websockets (s. [Fette and Melnikov(2011)]), welche die Realisierung einer bidirektionalen Verbindung über das HTTP ermöglichen.

Ein weiterer, sehr bekannter, Mechanismus verbirgt sich hinter dem Connection–Header. Über ihn kann definiert werden, wie die Kommunikationspartner mit der TCP-Verbindung umgehen. Hier findet sich ein entscheidender Unterschied im Standard–Verhalten der Kommunikation zwischen den HTTP–Versionen 1.0 und 1.1. Während im HTTP/1.0 pro Request eine TCP/IP-Verbindung erstellt und wieder geschlossen wurde, ist das Standard–Verhalten für HTTP/1.1 die Verwendung bzw. die Annahme einer persistenten Verbindung. Eine persistente Verbindung bedeutet hier die Verwendung einer TCP/IP–Verbindung über mehrere Requests hinweg. Dieser Mechanismus reduziert sowohl client– als auch serverseitig den Overhead beim Auf– und Abbau der Verbindung. Als weiteren Vorteil der persistenten HTTP–Verbindung ist das *Request Piplining* zu erwähnen. Dabei werden mehrere Requests aneinandergereiht und hintereinander über eine TCP/IP–Verbindung an den Server gesendet. Da das HTTP im Klartext Informationen austauscht, ist HTTPS eine Variante des Protokolls, bei der die Kommunikation verschlüsselt über die sogenannten Secure Socket Layer erfolgt. Das zählt jedoch, trotz der Ähnlichkeit im Namen, nicht zu den Fähigkeiten des HTTP.

2.1.4 HTML und HTML5

Wie der Name *Hyper Text Markup Language* (*HTML*) vermuten lässt, entstand *HTML* zusammen mit dem *Hyper Text Transfer Protocol* und dem *Unified Rescource Locator* am CERN. HTML ähnelt der *Extendet Markup Language*. Jedoch ist sie erst XML–konform in der als *XHTML* bekannten Version. Der vom W3C geprägte Begriff HTML5 umfasst nicht nur die aktuelle Version der *Hypertext Markup Language*. Er steht vielmehr als umfassender Oberbegriff für neue Web–Technologien. Das W3C fasst diese unter den folgenden Kategorien zusammen:

- Semantics
- Device Access
- Offline and Storage
- Connectivity (Websockets Server–Sent–Events)

- 3D & Effects Grafik im Browser 2D (HTML–Canvas) 3D (WebGL)
- Multimedia
- Performance and Integration
- CSS & Styling

2008 wurde vom W3C der erste *Working Draft* zum Thema HTML5 veröffentlicht. Der HTML5-Standard wird in einer Arbeitsgruppe geleitet und durch das W3C diskutiert. Zu dieser Gruppe zählen auch viele große Firmen wie Mozilla, Google, Microsoft, Apple, IBM, Opera, Nokia oder AOL, was eine hohe, weltweite Akzeptanz sicherstellt.

2.2 Das mobile WWW – vom Handy zum Smartphone

Die Anfänge des mobilen WWW finden sich in Techniken wie Wireless Application Protocol (WAP). Das sind speziell für mobile Geräte, wie beispielsweise Handys, optimierte Webseiten. Durch kleine Bandbreite, geringen Bedienkomfort und dem zusätzlichen Entwicklungsaufwand erfuhr die WAP Generation im europäischen Raum nicht die erwartete Aufmerksamkeit. Eine Ausnahme macht hier die MMS Technologie, welche auf der Kombination von SMS und WAP basiert.

Der Wechsel vom immer kleiner, leichter werdenden Handy zu Smartphones mit Bildschirmdiagonalen von bis zu 5,5 Zoll (13,97 cm) in Verbindung mit Touch–Bedienkonzepten und Breitbandanschlüssen, trieben die mobile Internetnutzung enorm voran. Gegenwärtig ist ein zunehmender Trend, weg von nativen Handy–Apps hin zu Plattform–unabhängigen Web–Apps zu beobachten. Dieser Trend wird durch die neuen API–Standards des HTML5 begünstigt. Sie haben das Ziel, Web–Apps die Nutzung spezieller Hardware– und Software–Features der jeweiligen Plattform zugänglich zu machen. Eine Übersicht zum Entwicklungsstand dieser Device–APIs ist unter [W3C()] der Device APIs Working Group des W3C zu finden. Beispiele dafür sind:

- *Media Capture and Streams* und *HTML Media Capture* API für die Nutzung von Audio– und Video–Schnittstellen eines Gerätes
- *Vibration* API erlaubt den Zugriff auf die Vibrationsfunktion des Gerätes
- *Calender* API für den Austausch von Terminen, Alarmen etc.
- *Geolocation* API (s. Geolocation Working Group des W3C)

2.3 Embedded–Systeme

Eingebettete Systeme (engl. embedded systems) sind kleine, hochspezialisierte Computer, die auf den speziellen Anwendungsfall zugeschnitten sind. Ihre hohe Spezialisierung auf Hardware–Ebene begünstigt eine hohe Performance bei vergleichsweise

geringen Ressourcen wie Rechenleistung, Strom– oder Speicherverbrauch. Durch ihre vielseitigen Einsatzgebiete besitzen diese Systeme keine oder nur sehr rudimentäre Schnittstellen für die Interaktion mit Benutzern. Allerdings hat sich der Begriffsumfang der eingebetteten Systeme in den letzten Jahren erweitert. So finden sich in vielen Bereichen des häuslichen und beruflichen (professionellen) Alltags Embedded–Systeme wieder. Die Palette reicht dabei vom Massenmarkt,hauptsächlich vertreten durch Consumer–Produkte wie DVD–Player, Kamera, Smartphones, MP3–Player, Bordcomputer im Auto, bis zu speziellen Systemlösungen in Industrie, Medizin, Wissenschaft, Luft– und Raumfahrt oder Militär. Beide Gruppen unterscheiden sich stark in den Anforderungen:

Wirtschaftlichkeit:	Produktionskosten, Wartungskosten und Unterhaltskosten
Leistung:	Performance/ Datendurchsatz eines Sytems
Wartbarkeit:	Handhabung von Software– und Hardwareupdates
Verfügbarkeit:	Erreichbarkeit bzw. Zugänglichkeit
Zuverlässigkeit:	Ausfallsicherheit, Stabilität
Sicherheit:	Absicherung vor unbefugtem Zugriff aber auch definiertes Verhalten im Fehlerfall bzw. Ausnahmesituationen

Die unterschiedlich stark ausgeprägten Anforderungen der beiden Bereiche Consumer und Professional wirken sich direkt auf die Wahl der Hardware und Software aus.

Der große Unterschied zwischen Embedded–Systemen für den Massenmarkt und für den professionellen Einsatz liegt in der Lebensdauer eines Produktes. Während Produkte im Consumer–Markt einen Lebenszyklus von einigen Monaten bis wenigen Jahren haben, sind Produkte für Industrie und Co. auf viele Jahre bis Jahrzehnte ausgelegt. Das bedeutet, es ist notwendig, auch noch in 10 Jahren Ersatzkomponenten und –systeme bereitstellen zu können.

2.3.1 Das Öffnen von lokalen Bussystemen hin zum Internet

Eingebettete Systeme sind im Vergleich zur Einführung des WWW noch nicht lange über das Internet oder lokale Netzwerke verfügbar. Erst durch die Bereitstellung von preisgünstiger Ethernet–Hardware kam es zur breiten Nutzung von Ethernet im Embedded–Umfeld. Ein erneuter Aufschwung kam durch „Drahtlose“ Technologien wie:

- Wlan
- Blutooth
- Near Field Communication
- mobile Breitbandanschlüsse

Anfangs wurden noch spezielle Computerterminals mit teurer Hardware für verschiedene Feldbussysteme (ProfiBus, ProfiNet, CAN, MODBus, RS485) benötigt. Mit dem

Einzug des Ethernets in die Steuerrechner der Industrieanlagen, können diese heute vom Arbeitsplatz aus einfach überwacht und konfiguriert werden. Spezielle Bussysteme werden nur noch für Sonderaufgaben verwendet. Die drahtlose Kommunikation ist die nächste Stufe von Konnektivität (Connectivity) im Embedded–Umfeld. Diese Entwicklungen haben starken Einfluss auf den Einsatz von Embedded–Systemen, sowohl im Bereich ihrer Anwendungs– und Aufgabenbereiche wie auch auf die an sie gestellten Anforderungen.

Ein aktuelles,kritisches Problem stellt die Sicherheit von Embedded–Systemen dar. Früher physisch getrennte und zueinander auf Protokoll–Ebene inkompatible Netzwerke sind zusammengewachsen und so potentiell von außen angreifbar. Damit erben Embedded–Applikationen nicht nur die Vorteile, sondern auch die Gefahren und Probleme des Internets.

2.3.2 Die Hardware

Embedded–Systeme kommen in unterschiedlichsten Hardware–Konfigurationen vor. Die Auswahl an möglichen Mikrocontrollern ist sehr breit gefächert. Für diese Arbeit stand Hardware der Firma Taskit zur Verfügung. Dabei handelte es sich konkret um die folgenden 3 Plattformen:

- PicosG20
- NanosG20
- SAMDIP-7X

2.3.3 Betriebsysteme

Für den Embedded–Bereich stehen eine Auswahl an verschiedenen Betriebssystemen bereit. Dazu zählen:

- spezialisierten Betriebssystemen wie:
  QNX, VxWorks, Nucleus, OSEK, OS-9, RTEMS
- spezielle eingebettete Versionen von Standardbetriebssystemen wie:
  Linux (Embedded Linux), NetBSD oder Windows (CE, XP Embedded, Automotive oder Embedded for PoS)

3 Web Services Architekturen

Die Definition eines Web Service des W3C ([David Booth(2004)]) lautet wie folgt:

Definition 3.1 A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Diese Definition geht davon aus, dass ein Web Service sich durch ein maschinenlesbares Format beschreibt und andere Systeme mit ihm über das *Simple Object Access Protcol (SOAP)* interagieren. Die Kommunikation mittels SOAP Nachrichten soll dabei über das HTTP stattfinden. Das HTTP ist ebenfalls ein maschinenlesbares Format und dank seiner weiten Verbreitung nahezu jedem internetfähigen Gerät bekannt. Die Definition für einen Web Service schreibt weiterhin vor, dass die Nachrichten im XML–Format in serialisierter Form übertragen werden. Auch wenn es mit *Web Application Description Language* (WADL) einen alternativen Service zu WSDL gibt, der nicht auf SOAP, aber auf HTTP basiert, so sind doch auch rein auf dem HTTP basierende Dienste durch das Protokoll (HTTP) selbst in einer maschinenlesbaren Form beschrieben (s. 2.1.3 ab Seite 8). Besonderheiten in der Verwendung des Interfaces eines Services werden über HTTP–eigene Mechanismen ausgehandelt. Auch die Festlegung auf das SOAP kann hier entfallen, da diese Zugriffe ebenfalls über die einheitliche Schnittstelle (HTTP–Methoden) des HTTP abgedeckt bzw. fest vorgegeben ist. Die Festlegung auf XML als Serialisierungskonzept schränkt die Kommunikationspartner im Web sehr ein. Denn nur jene die XML beherrschen, können diesen Service nutzen. Jedoch lassen sich leicht über den Accept bzw. Content–Type–Header des HTTP das für beide Seiten beste Format aushandeln.

Eine umfassendere Definition für einen Web Service könnte demnach wie folgt lauten:

Definition 3.2 Ein Web Service ist eine Maschine–zu–Maschine Kommunikations–-Software in einem interoperablen Netzwerk. Sein Interface wird durch das HTTP definiert. Andere Systeme kommunizieren mit dem Web Service wie es das HTTP vorgibt oder über ein vorher durch das HTTP ausgehandeltes Format. Nachrichten werden mittels HTTP–Nachrichten in einem von beiden Seiten über das HTTP vereinbarte Datenformat ausgetauscht.

Diese erweiterte Definition schließt auch RESTful Web Service in vollem Umfang mit ein, da der *Web Service* Begriff vorher einer Service–orientierten Architektur (SOA) in Verbindung mit dem *Simple Object Access Protocol* vorbehalten war. In der SOA

ist die Rede von Services. Demnach ist die Namensgebung Web Service, als Ableitung aus „über das Web interagierende Service“, hier intuitiv und verständlich. Bei der Ressourcen–orientierten Architektur (ROA) liegt der Fokus auf Ressourcen die via Web über eine einheitliche Schnittstelle verwendet werden können. Der Begriff Web Service bezeichnet hier also ein Programm, welches als HTTP–Server fungiert und Ressourcen im Web bereitstellt. Die beiden Architekturstile SOA und ROA sind zur Zeit die meist vertretenen Architekturkonzepte, wenn es um Bereitstellungen von Dokumenten bzw. Diensten im und über das Web geht. Sie werden im Folgenden kurz vorgestellt und auf ihre Verwendbarkeit im Bereich der Embedded–Systeme bewertet.

3.1 Einsatzgebiete

Wie schon einleitend in dieser Arbeit angedeutet, beschränkt sich das moderne Web nicht mehr nur auf den Austausch von Dokumenten. Vielmehr hat sich das WWW zu einer Informations– und Serviceplattform entwickelt. Mit dem sogenannten Web2.0 wurde eine neue Phase der interaktiven Nutzung des WWW eingeläutet.

Durch die Möglichkeiten des WWW wandelte sich die IT–Business–Landschaft. Die bisher genutzten *Legacy–Systeme* zur Abbildung geschäftskritischer Prozesse im Unternehmen, wandelten sich in der jüngsten Vergangenheit hin zur Öffnung und Vernetzung mit Systemen von Zulieferern und Kunden. Diese neuen, das WWW integrierende IT–Bussiness–Systeme, sind heute unter dem Begriff *Enterprise–Systeme* geläufig.

Die Embedded–Systeme unterlagen einem ähnlich großen Wandel, welcher bis zum Zeitpunkt dieser Arbeit noch fortschreitet. Mit der Erschließung des Web für Embedded–Systeme durch Kabel– und insbesondere durch drahtlose Technologien, steigt das Anwendungsfeld und damit die Nachfrage nach ihnen beständig. Im Consumer–Bereich schlagen sie in Form von Smartphones neue Anwendungsbrücken zu Enterprise–Systemen. Durch ihre Omnipräsenz im Alltag stellen sie den Service Endpoint der Enterprise–Systeme in Bezug auf den Kunden dar. Jeder kann zu jeder Zeit über alles informiert werden und als Kunde in Aktion treten. Im professionellen Bereich erhalten Konzepte, wie die des programmierbaren Web, mit Ansätzen wie *Web of Things* neuen Zulauf.

3.1.1 Enterprise

Enterprise–Systeme sind zur Verarbeitung (Reporting, Fakturierung) großer Datenmengen ausgelegt. Üblicherweise sind es Systeme, die viele Nutzer (> 10000) gleichzeitig bedienen. Die Entwicklung von Enterprise-Systemen brachte den konsequenten Schritt. . . von Fat Clients hin zu Thin Clients mit dem Ziel, die Datenhaltung sowie die Bussines–Logik zentral zu halten und zu verwalten. Diese Strategie folgt den Zielen

der Software–Architektur zur Vereinfachung von Wartung und Entwicklung. Der Aufwand für die Aktualisierung von Fat Clients in einem Softwaresystem dieser Größenordnungen ist äußerst schwer, zeit– und kostenintensiv. Der Entwicklung hin zu Thin Client ist hier der logischer Schritt, um den Aufwand der Implementierung und Wartung von Server–Diensten für verschiedene Client–Versionen zu vermeiden. Der Einsatz von Web–Technologien ermöglicht es, die Client-Applikationen weiter von Systemkomponenten zu befreien, die dem Lebenszyklus des Softwareprojektes unterliegen. Die Verwendung von generischen HTTP–Clients, wie beispielsweise Webbrowsern, erlauben es, Clientanwendungen mit der notwendigen Bedienlogik für das Nutzerinterface zentral zu verwalten. Die konsequente Zentralisierung von Datenhaltung, Bussineslogik und grafischer Nutzerschnittstelle, stellt besondere Anforderungen an das Enterprise–System.

hohe Skallierbarkeit: Das System muss viele Nutzer gleichzeitig bedienen können. Einzelne Hardware-Systeme sind dazu auch bei heutigen Systemresourcen nicht dazu in der Lage. Es ist hier notwendig, sich solcher Mechanismen wie *Load Balancing* zu bedienen.

hohe Ausfallsicherheit / Verfügbarkeit: Die Zentralisierung ist gleichzeitig der kritische Punkt eines solchen Systems. Ein Ausfall einer der drei Bereiche Datenhaltung, Bussineslogik oder grafisches Nutzerinterface beeinträchtigt das gesamte System. Hier kann durch eine ausreichende Redundanz der Subsysteme bzw. der Hardware entgegengewirkt werden.

Zur Realisierung dieser Anforderungen wird sich verschiedener Software–Techniken bedient:

Verteilte Programmierung:	RPC, RMI, JMS (CORBA, JAVAEE)
Multitasking/Multithreading:	Auslastung bzw. Nutzen von Mehrprozessor–Systemen
Middleware:	Auslagern und Verbergen von Teilen des Systems hinter Subsystemen (Persistenz–Framework, Application Server, Webserver), die autonome Mechanismen für die Lastverteilung und Hochverfügbarkeit besitzen.

Hieraus entstehen hohe Anforderungen an die Hardware:

- CPU-Leistung
- Arbeitsspeicher
- permanente Speicherkapazität
- Netzwerk– / Interconnectperformance

3.1.2 Embedded

Die Einsatzgebiete von Embedded–Systemen sind breit gefächert. Sie sind unter anderem zu finden in:

- Kommunikationstechnik
- Medizintechnik
- Gebäudetechnik
- Multimedia
- Automotiv
- Robotik
- ...

Zu ihren Aufgaben gehören im Allgemeinen das Messen über Sensoren und Interagieren mit ihrer Umwelt über Aktoren. Dabei zeichnen sie sich durch spezielle Rahmenbedingungen in ihrem Systemverhalten aus:

effizient: Eingebettete Systeme sind in der Regel in ihrer Hardware–Ausstattung für das spezielle Einsatzgebiet optimiert.

zeitlich deterministisch: Es muss garantieren können, Vorgänge in einer vorgegebenen Zeit zu einem Ergebnis zu bringen.

autonom: In der Regel soll ein System selbstständig, ohne Kontrolle und Einflussnahme eines Nutzer, arbeiten.

zuverlässig: Das System weist eine hohe Fehlertoleranz und Stabilität gegen Störeinflüsse auf.

sicher: Das System ist ausreichend gegen unbefugte Beeinflussung und Datenspionage geschützt.

Die Ausprägung der genannten Punkte unterscheidet sich je nach Anwendungsfall des Embedded–Systems. Zum Beispiel in einem Smartphone ist die Interaktion mit dem Benutzer die Hauptaufgabe. Demnach kann die Anforderung der Autonomie an das System sehr gering ausfallen. Eine weitere Anforderung ist eine hohe Akkulaufzeit, die ein energieeffizientes Gesamtsystem voraussetzt.

3.2 Architekturstile

Mit zunehmender Komplexität und Verwebung von einzelnen Programmen zu Prozessen und komplexen Prozess–Ketten wächst die Notwendigkeit, Abstraktionen von Problemen zu schaffen. Prozesse sind nicht mehr nur auf lokal verfügbare Ressourcen beschränkt, da sie mit anderen Prozessen und Ressourcen über Rechner– und Netzwerk–Grenzen hinweg interagieren. Prozesse könne als Dienst bezeichnet werden, wenn sie sich anderen Prozessen zur Erfüllung einer Aufgabe anbieten oder angeboten werden. Um ein solches Öko–System aus Ressourcen, Prozessen und Diensten überblicken zu können, hat sich die Analyse, Design und Implementierung auf verschiedene Abstraktionsniveaus eines solchen Systems durchgesetzt. So können die einzelnen Komponenten auf die externen Schnittstellen und die o.g. Anforderungen des Gesamtsystems abgestimmt werden. In der Gesamtheit wird ein solches System als Software–Architektur bezeichnet. Die Verwendung einer Software–Architektur hat zum Ziel (nach [**?**]) :

- das Entwicklungsprojekt effizienter gestalten
- Risiken minimieren durch frühe Berücksichtigung der Einflussfaktoren
- Verständnis schaffen bei allen Beteiligten
- Kernwissen über das System konservieren

Software–Entwicklungsmethoden verfolgen 3 grundlegende Ziele:

- Reduzierung von Kosten für Wartung und Entwicklung
- schnelle effektive Entwicklung (Time to Market)
- Erhöhung der Qualität der Software

Um die verschiedenen Komponenten einer Software–Architektur zu verknüpfen, gibt es verschiedene Techniken. Sie können in verschiedenen Formen vorliegen wie:

- Best Practise / Design Pattern / Philosophien & Paradigmen
- Protokolle und Standards
- Betriebsysteme, Frameworks, Libraries
- Benutzerschnittstellen und Bedienkonzepte

Ein Architekturstil ist eine spezielle Komposition dieser Mittel zur Umsetzung von Software–Architekturen. Architekturstile spiegeln deshalb häufig den momentanen Stand der Softwaretechnik (*State of the Art*) wider.

Die Wahl eines Software–Architekturstils sollte zu der Laufzeitumgebung passen bzw. alle Facetten des Projektes und seiner Umwelt erfassen bzw. zumindest mit einbeziehen können.

Die Zielarchitektur ist abhängig von :

- Systemressourcen
- verwendete / notwendige Schnittstellen
- Antwortzeiten

Die zur Zeit dieser Arbeit bekanntesten Architekturstile für Web Services sind Ressource Orientated Architecture (*ROA*) bzw. REST und Service Orientated Architecture (*SOA*).

3.2.1 Service Orientated Architecture

Wie der Name Service Orientated Architecure (SOA) schon sagt, ist das Grundkonzept dieses Architekturstils, alle Prozesse auf Services und Kompositionen verschiedener Services abzubilden. Diese Services bilden dabei untereinander hierarchische Strukturen. Services sind in unterschiedliche Prozess–Abstraktionsebenen vertikal untergliedert. Die vertikale Einordnung spiegelt die Komplexität und Verantwortlichkeit eines Services wider. Ein Service kann prinzipiell einen eigenen Prozess implementieren oder eine Komposition verschiedener Services zu einem neuen Prozess verbinden. Die SOA ermöglicht es sowohl Software–Architektur als auch die Komplexität realer Unternehmensprozesse in einen Kontext zu setzen. Diese starke Verknüpfung und die Abbildung von Geschäftsprozessen auf Systemprozesse sorgt für eine hohe Präsenz im Bereich der Enterprise–Systeme.

Ein sich an einer SOA orientierendes System kann nach [Horn(2008)] durch folgende Komponenten identifiziert werden:

- Anwendungs–Frontend
- Service Registrierung
- Service–Bus (intelligenter Service BUS wie BPMN, BPEL)
- Services

Ein Client[3] kann einen Dienst nicht direkt verwenden bevor er ihn bzw. seine „Adresse“ nicht kennt. Für die Vermittlung von Client und Service dienen die Komponenten Service–Registry und/oder ein Service–Bus. Diese werden von dem Client nach einem Dienst oder einer Kategorie von Diensten befragt. Ist ein auf die Anfrage passender Dienst vorhanden, bekommt der Client eine Referenz (Handel, Verbindung) zu diesem. Je nachdem wie intelligent die eingesetzte Registry– und Service–Bus Systeme sind, können Services nach folgenden Kriterien herausgesucht und zugewiesen werden:

[3] Anwendungs–Frontend oder ein anderer Service

- über einfache Namensauflösung (Namespace, URI, ÍD-String)
- nach Service–Typ mit speziellen Qualitätsmerkmalen (Antwortzeit, Authentifizierungsmechanismen, Datenformate, Preis)

Dieser Vorgang zur Ermittlung des Services oder des Service–Anbieters nennt sich Service-Discovery. Nach dem Finden eines geeigneten Services ist der Client nun in der Lage, sich mit diesem über das SOAP auszutauschen. Das Simple Object Access Protocol ist die verbreitetste Form zur Implementierung von serviceorientierten Architekturen. Es basiert auf XML und ist fast in jeder Programmiersprache, die Unterstützung für XML bereitstellt, anwendbar. SOAP ist ein eigenständiges Protokoll und kann unabhängig von der zugrunde liegenden Übertragungstechnologie verwendet werden, da es als Nutzlast transportiert wird. Im Umfeld der Web Services wird das HTTP für den Transport von SOAP Nachrichten verwendet.

3.2.2 Ressource Orientated Architecture

Der Architekturstil *Ressource Orientated Architecture* bedient sich des *Representational State Transfers* (REST) und entspricht damit der Natur des HTTP. Der Begriff REST wurde von Roy Fielding in seiner Dissertation [Fielding(2000)] geprägt um die Funktionsweise des WWW zu beschreiben.

Die Ressourcen in einer ROA stellen abstrakte Schnittstellen auf reale oder imaginäre Objekte dar. Beispiele dafür sind :

- die Außentemperatur
- der Gemütszustand einer Person
- das RFC2616

Während eine REST–Architektur sich auf die Eigenschaften des HTTP stützt, schließt die Ressourcen Orientierte Architektur auch andere Basis–Technologien, wie URI und Hypermedia nicht aus. Die Resourcen Orientated Architecture bildet den Oberbegriff für die Beschreibung eines Anwendungssystems, basierend auf dem Informationsaustausch über Ressourcen in einem komplexen Softwaresystem.

Der Representational State Transfer definiert die Art und Weise des Zugriffs auf Ressourcen, sowie deren Erstellung und Entfernung aus einem dem WWW ähnlichen System. Das HTTP stellt als Protokoll eine konkrete Implementierung für den Zugriff auf eine Ressource über ein Uniform Resource Identifier dar. Über die URI werden Ressourcen in einer solchen Architektur adressiert.

Das *Hypermedia*–Konzept bedient sich der Möglichkeit der Verlinkung von Ressourcen untereinander. Durch Verfolgen der Links, also Anfordern einer anderen Ressource über eine enthaltene URI, wird der Prozessfluss realisiert. Anders, als in einer SOA, ist hier

eine Servicebeschreibung in Form einer WSDL nicht notwendig, da alle Web Services die gleiche einheitliche Schnittstelle verwenden. Diese einheitliche Schnittstelle ist das zentrale Konzept der REST–Technik.

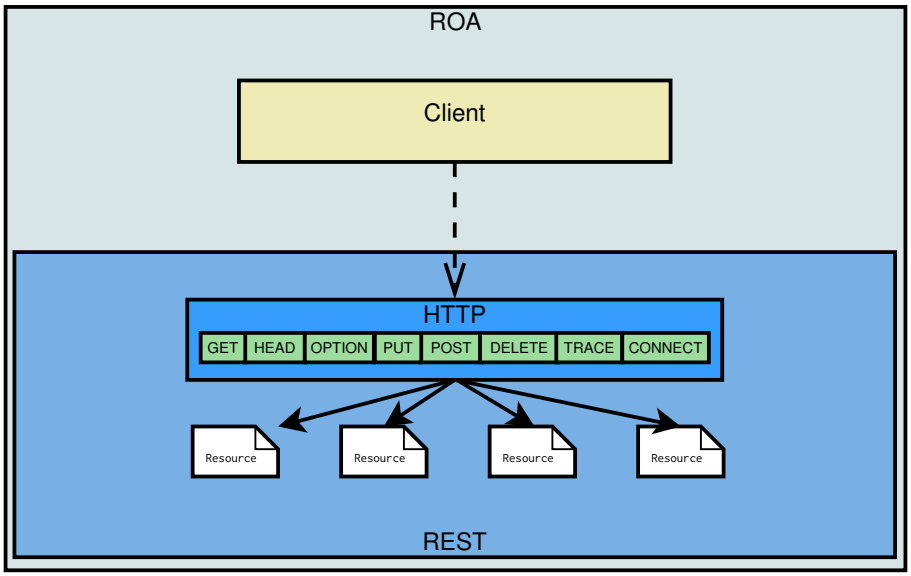

Abbildung 3.1: Die Abbildung zeigt eine hierarchische Einordnung der 3 Begriffe ROA, REST und HTTP zueindander.

3.3 Web Services in Embedded–Systemen

Mit der zunehmenden Vernetzung von eingebetteten Systemen und dem Wunsch nach mehr „Connectivity“, steigt hier das Interesse an etablierten Web Service Lösungen. Daraus lässt sich das Ziel ableiten, Embedded–Geräte ohne spezialisierte Hard– und Software bedienen zu können. So zum Beispiel soll ein Ingenieur in der Lage sein, aus der Stadt A –mittels Smartphone– seine Anlagen in der Stadt B, zu überwachen. Für die Lösung dieser Szenarios gilt dabei:

komfortabel: Komfortabel bedeutet hier ein einheitliches, einfach zugängliches Nutzer– und Programmier–Interface. Der Anwender soll mit einem Gerät viele verschiedene Systeme bedienen können. Dabei ist das verwendete Programm– und Bedienkonzept identisch. Die Synergien, welche durch ein intuitive Bedienung bzw. die einheitliche Schnittstelle von Ressourcen (RESTful HTTP) entstehen, führen zu einer Komfortsteigerung.

günstig: Der Einsatz von Standard Hard– und Software reduziert Entwicklungskosten, Unterhaltskosten und Lizenzkosten. Da bei der Wahl der Hardware auf verfügbare Massenprodukte zurückgegriffen werden kann. Bei der Softwareentwicklung kann auf das HTTP und darauf aufbauende frei verfügbare Middelware genutzt werden. Dies wird durch Lizenz-Modelle wie zum Beispiel GPL, BSD oder MIT möglich.

universell: Durch Verwendung von plattformübergreifenden Standards wird der Portierungsaufwand auf andere Systeme minimiert.

sicher: Der Forderung nach „Connectivity“ folgt der Ruf nach Authentifizierung und Autorisierung. Die Kehrseite der zunehmenden Vernetzung, ist die Anbindung und Zugänglichkeit von kritischen Systemen, z.B. Kraftwerke, Industrieanlagen mit Gefahrgut oder die Haussteuerung, an das Internet . Dafür stehen den Web Services verschiedene Authentifizierungs–Techniken des HTTP zur Verfügung.

3.4 Entscheidung für Embedded RESTful Web Services

Die Entscheidung für Embedded RESTful Web Services wird durch eine Gegenüberstellung der beiden Architekturstile ROA und REST in den folgenden Punkten entschieden:

- Ressourcenverbrauch
- Implementierungsaufwand
- Komplexität
- Kommunikationsoverhead

Der Ressourcenverbrauch kann anhand der einzusetzenden Software–Techniken gut abgeschätzt werden. Prinzipiell kommt für eine SOA basierte Lösung das SOAP zum Einsatz. Für die Realisierung über SOAP müssen sowohl HTTP als auch XML+SOAP implementiert werden. Eine RESTful Web Service–Implementierung beschränkt sich hier auf die reine Implementierung des HTTP. Dies ist mit deutlich weniger Systemressourcen zu realisieren als die SOA–Variante. Auch der Programmieraufwand selbst, stellt in einer SOA Lösung wesentlich höher Anforderungen. Die einfache Komplexität eines SOA–Systems ist für Embedded–System unverhältnismäßig hoch. Dagegen reicht für Zugriffe auf Basis des RESTful–HTTP die Netzwerk–Infrastruktur vollkommen aus.

In SOA–Lösungen müssen noch zusätzliche Komponenten wie Service–Bus oder Service–Registry gestellt werden, die in der REST Variante über vorhandene DNS–Dienste verfügbar sind. Die Tatsache, dass der eigentliche Serviceaufruf erst in einem SOAP als HTTP–Nachricht übertragen wird, spricht ebenso für die RESTful HTTP–Variante, bei der dieser, durch die im HTTP eingebauten Mechanismen, realisiert ist. Hinzu kommt

das XML als Datenaustauschformat im Vergleich zu JSON einen deutlich größeren Overhead hat.

3.5 Mögliche Embedded RESTFul HTTP Web Service–Infrastrukturen

Nach der Entscheidung für eine RESTful Web Service–Architektur bleibt die Wahl einer zur Anwendung passenden Netzwerk–Infrastruktur. Embedded Web Services können von den herkömmlichen Webserver–Infrastrukturen profitieren.

Die unterschiedlichen Varianten für die Umsetzung einer RESTful Web Service–Architektur bringen auch verschiedene Alternativen zur Nutzung von HTTP–Eigenschaften und Technologien mit. Es gilt die zur Wahl stehenden Varianten nach den Anforderungen des Projektes zu bewerten. Das sind:

- Skalierbarkeit
- Verfügbarkeit
- Ressourcenverbauch

Einige Infrastruktur–Konzepte lassen es zu, mehrere Web Services parallel zu betreiben und gleichzeitig eine Lastverteilung vorzunehmen. Das kommt zur Anwendung in Szenarien, in denen redundante Messungen erforderlich sind, um bei Ausfall eines Gerätes die kontinuierliche Datenerhebung zu sichern. Für einen Anwender ist es in einem solchen Fall nicht von Interesse, welches der Geräte seine Anfrage beantwortet. Bestehende Systeme können bei Bedarf einfach erweitert werden. Da für professionelle Embedded–Hardware eine lange Verfügbarkeit im Vergleich zu Consumer–Geräten besteht, können so die Entwicklungszeit und –kosten für ein neues System bzw. durch eine Portierung der Software auf eine neue Hardware eingespart werden. In der Umkehrung besteht die Option, ein System bei geringerer Last, ohne Beeinträchtigung des Gesamtsystems, zu reduzieren.

Eng verbunden mit der Skalierbarkeit ist die Verfügbarkeit. Muss sichergestellt sein, dass die erhobenen Daten jederzeit verfügbar sind, bietet es sich an, einen Web Service auf jedem dieser Geräte bereitzustellen. Der gleiche Mechanismus, der für die Lastverteilung sorgt, sichert so die Verfügbarkeit der benötigten Daten.

Einen weiteren Aspekt in der Entscheidungsphase für eine Infrastruktur stellt der Ressourcenverbrauch bzw. die Systemanforderungen der jeweiligen Lösung dar. Durch geschicktes Einbeziehen der Netzinfrastruktur kann die Belastung des *Embedded–Systems* stark reduziert werden.

3.5.1 FastCGI

Eine mögliche Infrastruktur basiert auf der Kombination eines Webservers und FastCGI. Nicht jeder Webserver eignet sich für Embedded–Systeme. Es gibt jedoch Webserver, die für geringen Ressourcenverbrauch entwickelt wurden.

Vorteile dafür sind:

- portabel unter Webservern mit CGI/FCGI Unterstützung
- große Freiheit bei der Wahl der Implementierung (Programmiersprache)
- viele Aufgaben werden vom Webserver abgenommen (Authentifizierung, Auslieferung von statischen Inhalten)
- Trennung von Webserver und FCGI über Systemgrenzen hinweg

Nachteil:

- Overhead bei der Kommunikation zwischen Webserver als Frontend und CGI/FCGI Programm als Backend
- bedingt tauglich für Mikrokontrollersysteme ohne Betriebsystem
- Abhängigkeit von einem FCGI–Unterstützenden Webserver

3.5.2 Standalone

Eine andere Strategie ist eine selbstständige Implementierung des Webservers.

Vorteile dafür sind:

- hohe Performance
- große Freiheit bei der Wahl der Implementierung (Programmiersprache)
- durch hohe Spezialisierung in der Implementierung breite Möglichkeiten in der Portierung

Nachteil:

- viele Funktionen des HTTP müssen implementiert werden: Authentifizierung, Auslieferung von statischen Inhalten
- hoher Implementierungsaufwand im Vergleich zu einer Webserver/FCGI basierenden Lösung
- hoher Testaufwand
- evt. hoher Aufwand in der Portierung

4 Datenrepräsentation

Während der Kommunikation über HTTP haben die Daten zu unterschiedlichen Zeiten unterschiedliche Strukturen . Diese richten sich nach den Stationen, die sie durchlaufen. Das Konsumieren von Datenrepräsentation ist nicht so aufwendig,wie das Produzieren. Das beeinflusst maßgeblich die Entscheidung für die eingesetzten Datenformate.

4.1 Datenrepräsentations-Schichten

4.1.1 Serverseitig

Serverseitig werden Daten erhoben und persistiert. Der Server übernimmt die Konvertierung der persistierten Daten in einem vom Client gewünschten Datenformat. Eine Beschränkung auf einfache Formate ist dabei üblich. Folgende Formate werden aufgrund der beschränkten Ressourcen verwendet: JSON,csv,text

4.1.2 HTTP–Nachricht

Im HTTP–Header wird das in der HTTP–Nachricht verwendete Datenformat vereinbart. Es gibt zwei Richtungen:

client → server: Der client übermittelt dem Server eine Datenrepräsentation mit einem Request. Dafür gibt er über den HTTP–Header *Content-Type* einen MIME-Type an.

server → client: Der Client übermittelt dem Server mit dem Accept–HTTP–HEADER, welche Dokumentenrepräsentationen er bevorzugt.

4.1.3 Clientseitig

Der Client muss die Fähigkeit besitzen, Daten aus der empfangenen Datenrepräsentation zu extrahieren oder eine abgehende Nachricht zu codieren. Der Client ist für die Präsentation, in einem für den Nutzer brauchbaren Format, zuständig.

4.1.4 Austausch und Transformation

Solange ein Web Service die benötigten Daten in der gewünschten Repräsentation anbietet, ist der Austausch trivial. Problematisch wird es, wenn der Embedded Web Service zum Beispiel aus technischen Gründen die benötigte Repräsentation nicht bereit-

stellen kann oder will. Für einen aktuellen Desktop–PC stellt eine Transformation der Daten auf Client–Seite heute kein größeres Problem dar. Im Browser steht mit JavaScript eine mächtige Script–Sprache zur Verfügung. Mithilfe dieser können Dokumentenrepräsentationen beispielsweise im JSON-Format vom Web Service angefordert werden und clientseitig mittels JavaScript–Bibliothek in ein PDF–Dokument transformiert werden. Diese Form der Auslagerung von Datenrepräsentations–Formaten aus der Datenrepräsentations–Domäne des Web Services in die des Clients, ist bei aktuellen Smartphones aufgrund deren Leistung möglich. Es gibt dennoch Fälle, in denen weder der Embedded Web Service noch der Client in der Lage ist, die benötigte Repräsentation zu erstellen. Mögliche Gründe dafür können sein:

Hardware Sowohl client– als auch severseitig sind nicht die benötigten Ressourcen für die Bereitstellung der Datenrepräsentation gegeben.

Lizenz Es wird ein proprietäres Format benötigt, dessen Erstellung aus lizenzrechtlichen Gründen oder fehlendem Know–how nicht möglich ist.

Eine mögliche Lösung währe die Verwendung eines „Datenrepräsentations–Proxy“. Dieser stellt ein Vermittler zwischen Web Client und Embedded Web Service dar. Er stellt Repräsentationen bereit, die von einem oder beiden Kommunikationspartner nicht unterstützen werden. Abbildung 4.1 zeigt seine Funktionsweise.

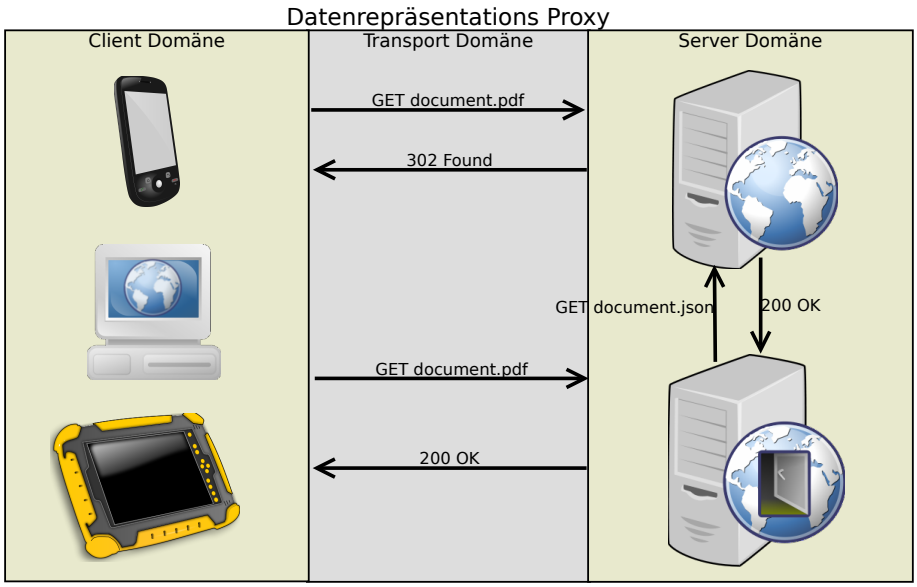

Abbildung 4.1: Ablauf der Kommunikation unter Verwendung eines Datenrepräsentations–Proxy. Der Proxy nimmt hier nur eine Transformation des Repräsentationsformates vor.

4.2 MIME–Types

Moderne Browser sind in der Lage, viele verschiedene Repräsentationsformate darzustellen. Ein MIME–Type ist eine Form, um Medien–Inhalte eindeutig zu identifizieren. Verwaltet werden sie von der IANA. Durch sie ist ein einheitlicher Ablauf zur Registrierung neuer MIME–Types festgelegt. Die MIME-Types sind in folgende Haupttypen unterteilt:

1. application
2. audio
3. example
4. image
5. message
6. model
7. multipart
8. text
9. video

.

MIME–Types müssen folgenden Eigenschaften genügen, beschrieben in Abschnitt 4 „Registration Requirements“, des [Freed and Klensin(2005)].

- Sie bestehen aus *type* und *subtype*.
- Sie sind case insensitive.
- Type und subtype dürfen maximal eine Länge von 127 Zeichen haben.

Es gibt eine sehr breite Palette an MIME-Types und damit möglichen Repräsentationsformaten für Daten eines Web Services. Für Embedded–Systeme ist es schwierig, eine breite Auswahl anzubieten, da das mit erhöhtem Speicherverbrauch durch entsprechenden Programm-Code einhergeht. Hier sollte eine sinnvolle, auf das Projekt zugeschnittene, Auswahl erfolgen. Grundsätzlich ist es aber ratsam, grundlegende MIME-Types wie text/plain oder text/html zu unterstützen. Ein Vorteil, der sich durch die Verwendung weit verbreiteter Formate ergibt, ist die größere Auswahl an bereits verfügbaren Parser Bibliotheken. Oft sind diese sogar schon für den Gebrauch in ressourcenkritischen Umgebungen optimiert. Jedoch ist bei besonders schmalen Bibliotheken darauf zu achten, dass sie mitunter nicht den vollen Umfang eines Formates unterstützen. Bei der Wahl der zu unterstützenden MIME-Types in Embedded–Anwendungen sollten die folgenden 3 Punkte ausschlaggebend sein:

1. Transport Overhead für die Information
2. Implementationsaufwand (schleifenlos, wenig puffern)
3. Selbstbeschreibend / Validierungsfähigkeit

Der Implementierungsaufwand beinhaltet sowohl die Komplexität zum Erzeugen der Ressourcen–Repräsentation als auch die notwendigen Systemressourcen. Parser Bibliotheken erhöhen schnell die Projektgröße, was ihren Einsatz bei begrenzter Kapazität des permanenten Speichers verhindern kann. Auch der RAM- und Rechenzeit-Verbrauch werden hier zu Ausschlusskriterien.

5 RESTful–HTTP Software Design

Für ein RESTful Web Service Design kann als Leitmotiv gelten:

Verwende das HTTP so, wie es ursprünglich gedacht war.

REST stellt kein neues Protokoll dar, sondern ist eher ein Rückbesinnen zum konsequenten Nutzen des *Hyper Transfer Protokolls* mit allen Features sowie scheinbaren Schwächen. Die grundsätzlichen Eigenschaften, die alle Ressourcen einer REST–Architektur gemeinsam haben, sind:

Verbundenheit	Ressourcen und deren Repräsentationen sind untereinander über URI verknüpft.
Adressierbarkeit	Jede Ressource in einer ROA hat mindestens eine URI zugewiesen.
Einheitliche Schnittstelle	Alle Ressourcen verfügen über die gleiche einheitliche Schnittstelle, bestehend aus den HTTP–Methoden.

Für die Erstellung eines Web Services muss ermittelt werden, welche Ressourcen das System besitzt und welche davon erreichbar sind. Ist die Frage der Ressourcen geklärt, steht die Wahl geeigneter Datenrepräsentationen an. Anhand einer Wetterstation soll exemplarisch dargestellt werden, wie die notwendigen Ressourcen aus einer realen Anwendung extrahiert werden können und welche Punkte bei der Bereitstellung der Ressource zu beachten sind. Der Wetter–Web–Service besteht aus einem PicosG20 als Hardware–Plattform und einem Kombi–Wettersensor zur Ermittlung von Wetterdaten.

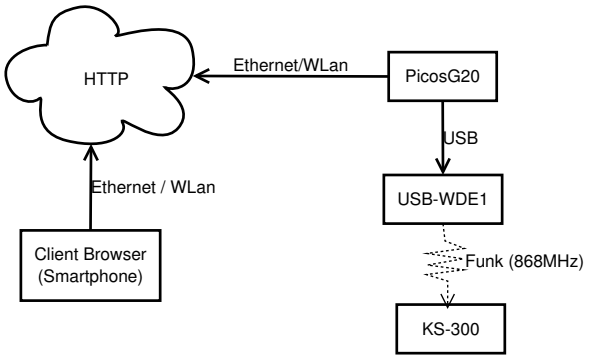

Abbildung 5.1: Blockschaltbild des Versuchsaufbau der Wetterstation

5.1 Ressourcen Design

Für die Identifizierung von Ressourcen ist es sinnvoll, zunächst die Kenngrößen des Systems zu ermitteln. Der Wetter–Web–Service ist über den Sensor in der Lage, folgende Daten zu ermitteln:

- Temperatur
- Luftfeuchtigkeit
- Regenmenge
- Windgeschwindigkeit

Solche Entwicklungen, an Standards vorbei, bergen die Gefahr in sich, die Vorteile durch die Nutzung von HTTP und des REST Architekturstiles zu untergraben. Das können, wi und auszuhebelne in [Leonard Richardson(2007)] oder [Leonard Richardson(2007)] beschriebene, leichte Inkonsistenzen in der Interface–Definition oder fatale Verhaltensfehlfunktionen des Webservers auf gestellte Anfragen sein. So zum Beispiel REST-RPC Hybride, welche in einem GET–Request Funkionsaufrufe der Form `/?action=delete&item=0815` beinhalten, damit die Idempotenz der GET–Methode verletzen und eine womöglich ungewollte Aktion auf eine Ressource ausführen. Dabei ist zu bedenken, dass im heutigen WWW nicht nur Anwender mit Webbrowsern als HTTP-Client auf den Web Service treffen, sondern auch Crawler oder andere automatisierte Programme, welche sich der Informationen im WWW bedienen.

5.1.1 Identifikation von Ressourcen

Der erste und wichtigste Schritt, ist wie in jeder Architektur, die Identifikation der systemrelevanten Größen. Dazu muss die Frage nach den verfügbaren Ressourcen, deren Herkunft und Format geklärt werden. Bei der Wetterstation werden über eine einfache serielle Schnittstelle Temperatur, Luftfeuchtigkeit, Windgeschwindigkeit und Regenmenge im *OpenFormat* ausgelesen. Aus diesen real verfügbaren Daten, lassen sich noch weitere Daten ermitteln, die als Ressource von Interesse sind:

- Eine Ressource, die alle verfügbaren Wetterdaten in sich vereint.
- Die höchste gemessene Temperatur.
- Die durchschnittliche Temperatur über einen bestimmten Zeitraum.
- Temperatur des Vortages

Bei der Auswahl dieser zusammengesetzten Ressourcen mussten die Rahmenbedingungen des Embedded Systems beachtet werden. Besonders bei algorithmische Ressourcen, die sich auf „Aggregierungs“–Funktionen wie MAX, MIN, AVG oder SUM zurückführen lassen, ist Sorgfalt geboten. Diese Funktionen können bei größeren Datenmengen schnell eine hohe Auslastung verursachen. Eine weitere Möglichkeit zur Iden-

tifizierung der notwendigen Ressourcen, kann im Umkehrschluss bedeuten, dass aus speziellen Datenrepräsentationen die benötigten Ressourcen abstrahiert werden können. Ist zum Beispiel vorgegeben, dass ein Client eine JSON–Repräsentation mit den Temperaturen der letzten 10 Tage benötigt, kann daraus eine Ressource *temperature/last10days* abgeleitet werden. Zu beachten bei dieser Methode ist, dass die Ressourcen vom MIMETYPE befreit wird. Das bedeutet, eine Ressource sollte unabhängig vom späteren Repräsentationsformat definiert werden.

5.1.2 Design von Ressourcen

In dem RESTful Web Service–Konzept nehmen Ressourcen eine führende Rolle ein. Sie auszuliefern bildet den Mittelpunkt eines jeden Web Services. Demnach ist das Design der Schnittstelle für den Zugriff über HTTP genauso wichtig, wie das Design der Ressourcen–Repräsentation. Zum Entwerfen einer Ressource müssen folgende Punkte geklärt werden:

- Unter welchem Unified Ressource Identifier soll die Ressource erreichbar sein?
- Welche Methoden der einheitlichen HTTP–Schnittstelle benötigt die Ressource?
- Welche möglichen Fehler–Konditionen gibt es?
- Mit welchen HTTP–Status–Codes soll reagiert werden?

Folgende Fragen müssen nach der Definition der Ressourcen überprüft werden:

- Sind ausreichend granulare Zugriffe auf die Ressourcen realisierbar?
- Sind alle nötigen Aspekte der Ressource vorhanden bzw. erreichbar?
- Gibt es Gemeinsamkeiten oder Überschneidungen zwischen Ressourcen?

.

5.1.3 Design der URIs

statische URIs

Statisches Routen einer URI zum verantwortlichen Controller ist einfach und mit geringem Ressourcenverbrauch zu realisieren. Hier können Datenstrukturen, wie z.B. Bäume zum Einsatz kommen.

`/subsystem/config` – *SubsystemConfigController*
`/subsystem/devices/config` – *DeviceConfigController*

dynamische URIs

Dynamisches Routen der URIs gestaltet sich aufwendiger. Sie beinhalten eine oder mehrere variable Elemente in der Pfadhierarchie. Sie lassen sich durch folgende Eigenschaften charakterisieren:

- mindestens ein variabler Pfadabschnitt
- potenzielle *n* Möglichkeiten pro Pfadvariable

Beispielhafter Aufbau einer URI mit variablen Anteil und Zuweisung zu einem Controller:

`/subsystem/{subsystemname}/config` – *SubsystemConfigController*
`/subsystem/devices/{devicename}/config` – *DeviceConfigController*

5.2 Die Fähigkeiten des HTTP und der Nutzen für Embedded–Systeme

Die Verwendung von RESTful HTTP für einen Web Service standardisiert eine Vielzahl an Mechanismen für typische Probleme. So zum Beispiel:

- Autorisierung und Authentifizierung
- Reduzierung der Kommunikation (Caching,Konditionales GET, Nachrichtenkompression)
- Protokoll–Erweiterungen (WebSockets, Tunneling)

5.2.1 Autorisierungs– und Authentifizierungs–Mechanismen

Der HTTP–Standard definiert zwei Authentifizierungsmechanismen, Basic– und Digit–Authenticiation. Diese einfachen Authentifizierungsmethoden werden von allen gängigen HTTP–Servern unterstützt.

Basic Authentication

Dieser Mechanismus ist der primitivste und anfälligste. Er basiert auf der einfachen Verkettung von Nutzername und Passwort. Der mit einem „:„ verbundene String wird Base64 encodiert übertragen. Da Base64, anders als Hash-Funktionen wie z.B. *MD5*, eine Umkehrfunktion besitzt, ist diese Methode sehr unsicher und sollte nur über eine verschlüsselte Verbindung wie SSH oder SSL verwendet werden.

Digit Authentication

Die Digit–Methode bietet mehr Schutz vor dem Passwortdiebstahl, da hier das Passwort nur zur Erzeugung eines temporären Schlüssel genutzt wird und als solches nicht in der Kommunikation auszuspähen ist. Der Digist–Authentication–Mechanismus ist in [Franks et al.(1999)Franks, Hallam-Baker, Hostetler, Lawrence, Leach, Luotonen, and Stewart] beschrieben.

Weitere Verfahren

Es gibt eine Reihe weiterer Systeme, die sichere Authentifizierungen mit granularen Autorisierungsmechanismen erlauben. Diese benötigen jedoch zusätzliche Infrastruktur und sind somit für Embedded–Systeme nur bedingt geeignet. Beispiele dafür sind Systeme wie Kerberos und openAuth.

5.2.2 Reduktion des Kommunikationsaufwandes

Im Embedded–Umfeld können die im Folgenden genannten HTTP–Funktionen maßgeblich zur Entscheidung der Durchführbarkeit von Embedded Web Service Projekten beitragen. So ist es in diesem Bereich häufig nötig, mit dem Transfer–Volumen und der Bandbreite der Internetverbindung hauszuhalten.

Content–Encoding

Das HTTP Protokoll bietet seit der Version 1.0 die Möglichkeit zu Komprimierung. Die Komprimierung der Nutzlast bei der Kommunikation ist im herkömmlichen Internet weit verbreitet. Sie reduziert den Traffic eines Internetdienstes und entlastet so den Server, den Client und die dazwischenliegende Netzwerk–Infrastruktur. Das bedeutet Kostenersparnis, da der Datentransfer trotz „Flatrate“ und „Traffic inclusive“, immer noch Geld kostet.

Über die den HTTP–Header Accept–Encoding kann der Client dem Server mitteilen, welche Kompressionsmechanismen er versteht. Der Server wiederum gibt durch den HTTP–HEADER Content–Encoding die verwendete Kompressionsmethode für die HTTP–Entity an.

Persistente Verbindungen und Request Pipelining

Das HTTP setzt auf dem TCP/IP Protokoll der OSI Ebene 5 auf. Ein HTTP Request zieht so ein TCP/IP Verbindungsaufbau mit zugehörigem Handshake nach sich. Ein

solcher Verbindungsaufbau kann je nach Netzauslastung und Entfernung einige Millisekunden in Anspruch nehmen. Auch fallen bei jedem Verbindungsaufbau Daten an. Der Verbindungsaufbau kostet somit Zeit und Transfervolumen. Dieser kann für einen einzelnen Request zwar durchaus vernachlässigt werden, jedoch stellt ein Client, der dem Hypermedia–Gedanken des RESTful HTTP folgt, nicht nur eine Anfrage an den Server. Zur Entlastung wurde mit dem HTTP 1.1 Standard persistente Verbindungen und Requestpipelining eingeführt. Diese Mechanismen dienen im herkömmlichen Internet der Entlastung und zur Einsparung an Ressourcen. Ab der Version 1.1 wird jede HTTP–Verbindung vom Server als persistent angenommen, es sei denn, es wird explizit über den Request Header *Connection: close* eine nicht persistente Verbindung gefordert. Der Wunsch nach einer persistenten Verbindung wird ebenfalls über den Request Header *Connection* mitgeteilt. Request Pipelining beschreibt einen Mechanismus, bei dem über eine Verbindung mehrere HTTP–Requests hintereinander gesendet werden können , ohne dabei auf die Beendigung der vorherigen Requests warten zu müssen. Die HTTP–Responses werden in der selben Reihenfolge beantwortet, wie die Requests eingegangen sind.

Caching und Konditionales GET

Über die Caching Funktionalitäten des HTTP kann vermieden werden, dass sich selten oder nicht ändernde Datenrepräsentationen wiederholt erstellt werden müssen. Das sogenannte *Konditionale GET* ermöglicht es dem Client, dem Server mitzuteilen, dass er nur eine neue Repräsentation erhalten möchte, wenn sich diese seit seinem letzten Zugriff geändert hat. Für diese Technik wird der HTTP–Header ETAG genutzt.

Der Client ist in der Lage dem Server mitzuteilen, welchen Auszug einer Repräsentation er benötigt. Das wird durch die Angabe des *range:xxx-yyy* Meta-Tags realisiert. Mit Hilfe dieser Funktionalität des HTTP, ist es möglich, redundanten Traffic einzusparen. So zum Beispiel mehrfaches Senden einer kompletten identischen Repräsentation, wegen einer instabilen Verbindung während der Datenübertragung. Beispielsweise ist ein Client in der Lage von einem 1MB großen Dokument nur den Teil vom Server anzufordern, welcher zuvor nicht mehr übertragen wurde. Denkbar wäre auch eine Situation, in der ein Client gezielt nur den gewünschten Teil einer Ressourcen–Repräsentation anfordert. Zum Beispiel bietet ein Web Service das Log eines Messgerätes an. Besteht ein Logeintrag nun immer aus einer festgelegten Länge an Zeichen und die Logeinträge werden zu einem festen Intervall angelegt, kann sich der Client gezielt mittels des *range* Meta-Tags den gewünschten Logauszug anfordern. Von dieser Möglichkeit zur Einsparung von Transfer–Volumen können allerdings nur Clients Gebrauch machen, welche die nötige Intelligenz und das Wissen um die Beschaffenheit der Datenrepräsentationsformate mitbringen. Es handelt sich also in diesem Fall immer noch um eine RESTful–HTTP Umsetzung. Clients ohne dieses Wissen werden dementsprechend mehr Traffic auslösen. Dennoch sollte vor dem Einsatz einer solchen Methodik das Design der Ressourcen noch einmal nach 5.1.2 überdacht werden.

6 Typen von Client–Applikationen

Client–Applikationen bilden die Schnittstelle zum Bediener (human machine interface - HMI) und zur Serverwelt der Web Services. Sie sollen nach Möglichkeit in ihrer Bedienung durch den Anwender sowie in ihren Programmierschnittstellen plattformunabhängig ausgeführt sein und einem applikationsübergreifenden Konzept folgen. Beispiele hierfür sind Gnome Desktopenvironment für HMI und D-BUS für Programmierschnittstellen. Das garantiert bei der Softwareentwicklung für unterschiedliche Plattformen einen hohen Grad an Wiederverwendung.

6.1 Web–App

Web–Apps sind Anwendungen, die in einem Browser ausgeführt werden. Vorteile bei der Entwicklung einer Web–App sind:

- einfache Implementierung der GUI (HTML, JavaScript)
- plattformunabhängig (höchst mögliche Portabilität)
- Look and Feel einer Nativen–App

Web–Apps haben gegenüber Nativen–Apps einige Einschränkungen . So können sie nicht auf Soft– und Hardware der jeweiligen Plattform zugreifen. Allerdings wird sich hier in Zukunft durch die Entwicklung von API Standards der Abstand zu Nativen–Apps verringern.

6.2 Native–App

Native–Apps sind speziell für die jeweilige Plattform entwickelte Anwendungen. Sie werden in der Regel mit einem vom Hersteller angebotenen SDK entwickelt. Dabei sind zum Beispiel die Programmiersprache, das Look & Feel,verwendbare Bibliotheken und Schnittstellen vorgegeben. Daraus ergibt sich auch der schwerwiegendste Nachteil nativer Apps. Sie sind für jede Plattform speziell zu entwickeln. Mitunter müssen in einem Programm Anpassungen für verschiedene Modelle eines Herstellers vorgenommen werden. Das stellt sich bei der Lebensdauer und Typenvielfalt von Smartphones als ein aufwendiges und somit teures Unterfangen dar. Es gibt hier Abhilfe durch einige Frameworks und IDEs, welche eine Portierung von Anwendungen auf unterschiedliche Plattformen möglich machen sollen. Diese sind aber auch wieder nur zu einer Auswahl von Herstellern und Modellreihen kompatibel. Häufig werden zum Erstellen einer Applikation trotzdem noch herstellerspezifische Bibliotheken benötigt. Hinzukommt eine weitere Abhängigkeit des Projektes von der Pflege und Aktualisierung des verwendeten

Frameworks und der Entwicklungs–IDE. Dem gegenüber bieten native Apps Zugriff auf die Hard– und Software der Plattform. So können sie zum Beispiel:

- eine vorhandene Kamera benutzen
- Sensoren wie Gyroskop, Höhenmesser, Helligkeits–, Annäherungs– oder Beschleunigungssensoren abfragen
- mit Systemprogrammen kommunizieren, wie dem messenger client oder den multimediaplayer
- einschalten des Vibrations Alarmes.

6.3 Hybrid–App

Ein Mittelweg aus Web–App und Native–App bildet die Gruppe der Hybrid–App. Sie versucht die Vorteile beider App–Typen in sich zu vereinen. Die Idee ist eine Zweiteilung in :

WebContainer für die Darstellung und Bedienung.

PlattformApp für die Realisierung des Zugriffes auf plattformspezifische Dienste (Sensoren, Kamera, . . .)

Die Nutzung des Webcontainers erlaubt die Umsetzung der GUI plattformübergreifend mit Webtechniken wie HTML5, CSS und JavaScript(AJAX). Die Abspaltung der Oberfläche aus der Nativen–App erlaubt es, sie wie eine gewöhnliche Webanwendung aus dem Netz zu laden.

Ein Beispiel für eine Multiplattform–Entwicklungsumgebung ist die Cross–Plattform Entwicklungsumgebung für die Entwicklung von Hybriden Applikation. Es können Apps wahlweise in C/C++ oder HTML5/Javascript geschrieben werden.

6.4 Entscheidung und Perspektiven

Die Entscheidung zwischen den Varianten ist sehr projektabhängig. Das trifft im Internet, im Enterprise Bereich sowie auch für die Realisierung von Anwendungen zur Steuerung von Embedded–Systemen gleichermaßen. Es kommt entscheidend darauf an, welche Funktionalitäten auf der Client–Seite benötigt werden.

Speziell bei Embedded Web Services muss zusätzlich überprüft werden:

- Leistungsfähigkeit der geplanten Client–Geräte
- Netzanbindung des Embedded Web Service (Kapazität, Volumen, Preis)

Prinzipiell gilt, dass alle Varianten in der Lage sind, HTTP zu nutzen und damit als

Web Client zu fungieren. Etwaige Sicherheitskonfigurationen einiger Plattformen, die ein solches Verhalten aus unterschiedlichsten Gründen verbieten, sollen hierbei ignoriert werden, da diese aus technischer Hinsicht dazu in der Lage sind.

7 RESTful Web Service Library (librws)

Mit der *RESTful Web Service Library* sollen die theoretischen Überlegungen der Entwicklung von RESTful Web Services auf Embedded–Geräten eine praktische Umsetzung erhalten. Dabei stehen folgende Ziele im Vordergrund:

1. hohe Wiederverwendbarkeit und Portierbarkeit
2. reduzieren der Einarbeitungszeit für Wartung und Entwicklung
3. Umsetzung von RESTful Web Services erleichtern
4. Integration in laufende und bestehende Projekte

Diese Ziele ergeben sich aus den vorgefundenen Konzepten für die Erstellung von Web Services. Die Wiederverwendung von Web Service–Komponenten wurde häufig durch Kopieren von Quellcode realisiert. Dieser musste anschließend zeitaufwendig angepasst werden. Ein Austausch unter einzelnen Entwicklern endete so häufig in einer eigenen oder separaten Weiterentwicklung der gleichen Problemstellung. Getestete Software–Komponenten kamen so meist nur in einem System zum Einsatz. Das Fehlen einer einheitlichen Strategie in der Planung und Implementierung von Web Services machte es sehr aufwendig, bestehende Software–Komponenten in anderen Projekten weiter zu verwenden. Das zeigte sich insbesondere bei der Portierung auf andere Hardware–Plattformen. Auch die Wartung gestaltet sich sehr ineffizient, da meist nur ein Entwickler mit dem System vertraut ist und andere sich erst zeitaufwendig in das System einarbeiten müssen. Die 3. Zielstellung ergab sich aus dem Wunsch, die Vorteile einer REST–basierenden Web–Schnittstelle auch für kleine Projekte zu nutzen, ohne den zusätzlichen Entwicklungsaufwand stemmen zu müssen. Das letzte Ziel ergibt sich daraus, nicht nur in zukünftigen Entwicklungen auf eine REST–Architekur zu bauen, sondern auch frühere und laufende Projekte dahingehend zu erweitern. Vorgegeben ist die Verwendung des Webservers lighttpd in Verbindung mit FCGI.

7.1 Konzept

Um Lösungsansätze für die Realisierung ermitteln zu können, sind zunächst die Anforderungen an die Bibliothek aus den Zielen zu extrahieren und näher zu spezifizieren. Anschließend können aus ihnen Konzepte zur Lösung der Aufgaben entwickelt werden. Um eine hohe Portier– und Wiederverwendbarkeit zu erreichen, sind die folgenden Punkte zu klären:

- einheitliche Vorgabe für die Entwicklung von Web Services
- generalisieren wiederkehrender Prozess–Strukturen in Web Services
- Identifizieren und Abstrahieren der Plattformabhängikeiten

Die Reduzierung des Wartungs– und Entwicklungsaufwandes kann erreicht werden durch:

- Wiederverwendung getesteter Software–Komponenten.
- Intuitive Programmierung durch projektübergreifende Software–Richtlinien.

Um den Entwickler in der Realisierung von RESTful Web Services zu unterstützen, muss die Bibliothek eine Struktur vorgeben, welche den Entwicklungsprozess hinsichtlich einer RESTful–Webschnittstelle leitet, ihn aber nicht einschränkt oder dominiert. Für eine Integration in bestehende und laufende Projekte müssen bei der Planung und Entwicklung die Rahmenbedinungen der Zielplattformen sowie vorhandene Konzepte berücksichtigt werden.

7.1.1 Plattform–Abstraktionen

Für eine höchstmögliche Portier– und Wiederverwendbarkeit muss die Bibliothek den Web Service weitestgehend von der zugrunde liegenden Plattform entkoppeln. Für diese Abstraktion können drei Ebenen festgehalten werden:

- Hardware–Ebene
- Betriebsystem–Ebene
- Laufzeitumgebung–Ebene

Jedoch ist es nicht möglich und auch nicht notwendig, für alle dieser Ebenen eine gleichermaßen umfassende Abstraktionsschicht durch die *librws* zu realisieren.

Hardware–Abstraktion

Eine Abstraktion der Hardware durch die Bibliothek kommt aufgrund der Notwendigkeit der hardwarenahen Programmierung in Embedded Web Services nicht in Frage. Eine Portierung von Web Services mit hardwarespezifischen Zugriffen, muss über externe Komponenten erfolgen. Ein Beispiel hierfür ist das */proc*-Filesystem. Allerdings ist dieses nur unterhalb Linux–basierenden Systeme anwendbar. In jedem Fall muss als Paradigma sowohl für die Bibliothek selbst, als auch für alle -sie nutzenden- Web Services gelten:

- verzichten auf spezifischen Code in Web Service–Modulen
- trennen von Hardware-Zugriffen über austauschbare Software–Module vom Web Service

Betriebsystem–Abstraktion

Betriebsystemspezifische Schnittstellen bilden ebenfalls eine Hürde, die es bei Portierungen zu bewältigen gilt. Eine konsequente Nutzung des *Portable Operating System Interface* (POSIX) kann viele dieser Hürden schon im Voraus ausschließen. Aufgrund der überschaubaren Menge an eingesetzten Betriebssystemen (Linux, NutOS, WinCE/Embedded), kann die Bibliothek einen kleinen Satz an Basisfunktionalitäten abstrahieren. Dafür ist ein einheitliches Interface zu definieren und für jedes zu unterstützende System in einem separaten Modul zu implementieren. Ein Beispiel für ein solches Modul ist die Vereinheitlichung zur Verwendung einer seriellen Schnittstelle unter verschiedenen Betriebssystemen. Um in allen Web Service–Implementierungen, diese serielle Schnittstelle verwenden zu können, ist ein simples Interface, das einen Satz von fünf Funktionen zum

- Öffnen
- Schließen
- Lesen
- Schreiben
- Konfigurieren

definiert, völlig ausreichend.

Laufzeitumgebungs–Abstraktion

Ziel für die Abstraktion der Laufzeitumgebung ist es, die zugrunde liegende Middelware wie Webserver,
glsglos:CGI-Varianten, mit ihren unterschiedlichen Schnittstellen auf eine einheitliche Schnittstelle abzubilden. Für diese Arbeit ist maßgeblich das
glsglos:FastCGI–Interface von Interesse. Weitere mögliche Laufzeitumgebungen sind *glsglos:NutOS–Webserver* und der *Mongoose*–Webserver (s.h. Abbildung 7.1 S. 46).

7.1.2 HTTP/REST Automatisierung

Eine automatische Nutzung von HTTP Technologien nach der REST Philosophie sowie eine einfache klare Struktur des Programms als Vorgabe, soll es dem Entwickler ermöglichen, sich auf die wesentlichen Aspekte des Web Service zu konzentrieren. Während der Entwickler für Implementierung der Bussines–Logik zum Zugriff und Aufbereitung der Daten verantwortlich ist, soll die Bibliothek im Hintergrund automatisch HTTP–Features erkennen und den Repsponse dahingehend modifizieren. Mögliche Ansatzpunkte dafür sind:

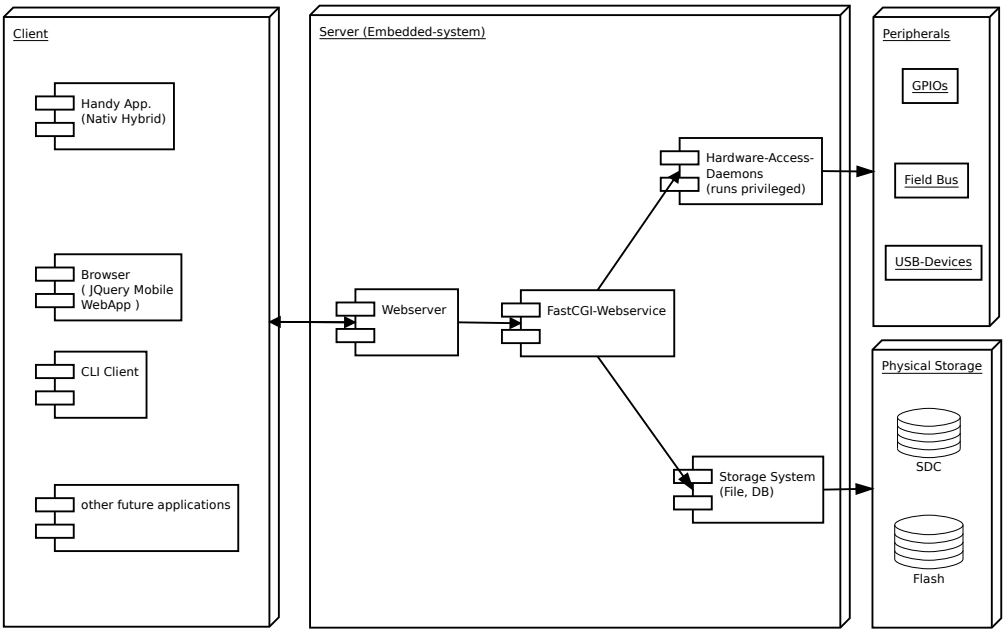

Abbildung 7.1: Domain Model eines Web Service mit librws

- automatisches Ermitteln unterstützter Kompressionsformate
- automatische De-/und Komprimierung der HTTP–Entity
- selbstständige Ermittlung und Bereitstellung des favorisierten MIME-Types
- automatische Generierung von Fehler–Responses (z.B. 405 METHODE NOT ALLOWED)

7.1.3 Integration

Damit sich die Bibliothek optimal in Buildsysteme bzw. in firmeninterne Softwareentwicklungsabläufe integriert, müssen folgende Vorgaben einbezogen werden:

- Kompatibilität zum openembedded Build–System
- Kompatibilität zu vorhandenen Softwarekomponenten, wie der *libtrie* oder *tatskit-modbus-library*

7.2 Design

Die Bibliothek wird in der Programmiersprache *C* implementiert. Um Ziele, wie die einfache Programmierung und Wartbarkeit zu realisieren, wird ein, an die objektorientierte Programmierung angelehntes Modell, in *c* nachgebildet.

Die Funktionsweise eines Web Services mit der *librws*basiert auf einem Pattern.

Die HTTP–Kommunikation wird über *HttpRequest*– und *HttpResponse*–Objekte reali-

siert. Die eigentliche Applikationslogik wird in Controller–Klassen vom Entwickler implementiert. Die Controller werden an URI gebunden und über einen Routing–Mechanismus angesprochen. Die Abbildung einer HTTP–Ressource bzw. eines HTTP–Ressourcentyps erfolgt idealerweise auf eine Controller.

7.2.1 Architektur

Für die *librws*bietet sich eine 4-Schichten–Architektur an, um die Forderung nach Abstraktion und Portierbarkeit zu erfüllen. Dabei werden folgende 4 Ebenen eingeführt:

1. Application Layer
2. Library Interface
3. System oder Runtime Abstraction Layer
4. System- / Device-Layer

Ziel dieser Stapelung ist es, klare Trennlinien zwischen einzelnen Funktionskomponenten zu ziehen. Durch diese Zerlegung bzw. Entkopplung der gesamten Request-Response Aktionskette, ist es möglich, Interfaces für Funktionsblöcke festzulegen. Als Funktionskomponenten oder –blöcke werden die einzelnen Aufgaben in der Verarbeitung eines Http– Requestes angesehen.

System– / Device– Layer

Die unterste Ebene gehört nicht mehr zur *librws*. Sie stellt die Schnittstelle zur Plattform dar, auf der die *librws*aufsetzen kann. Folgende drei Schnittstellen sind denkbar:

1. ein Framework wie FCGI
2. ein Betriebssystem (POSIX–Schnittstelle)
3. direkter Hardware–Zugriff

Runtime Abstraction Layer

Der Runtime Abstraction Layer abstrahiert aus dem System– / Device–Layer das für die Kommunikation benötigte Interface. Es genügen die drei Funktionen

- printf()
- write()
- read()

Weiterhin befindet sich hier der plattformabhängige Teil zum Starten, Stoppen und Managen eines *librws*–Web Services. Der Runtime Absraction Layer stellt über drei Klas-

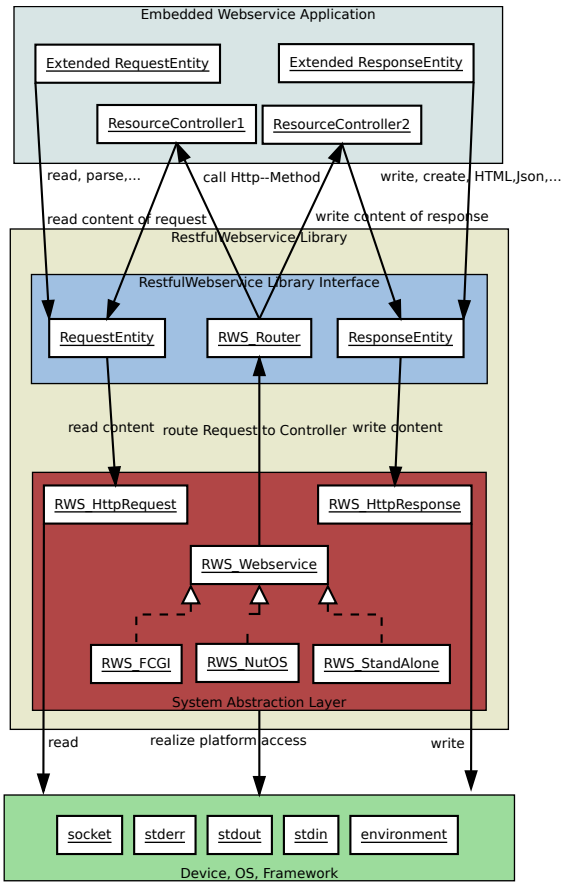

Abbildung 7.2: Architektur Konzept der librws für einfachere, portierbare RESTful Web Services auf Embedded– Systemen

sen alle notwendigen Schnittstellen für die darüberliegende Ebene Libraray–Interface.

RWS_Web Service Die Klasse RWS_Web Service stellt den Kern eines *librws*–Web Services. Sie beinhaltet die Hauptschleife für die Verteilung und Bearbeitung von eingehenden Requests. Über sie kann der Web Service kontrolliert werden.

RWS_HttpResponse Die KLasse RWS_HttpResponse erlaubt eine einfach Manipulation der Response Header über einfache getMetaTag– und setMetaTag– Funktionen. Sie verwaltet weiterhin sämtlich Schreibzugriffe für einen HTTP–Response an einen Client.

RWS_HttpRequest Mit Hilfe der Klasse RWS_HttpRequest werden die Lesezugriffe auf einen Message–Body eines HTTP–Requests von der Laufzeitumgebung abstrahiert. Sie ermöglicht außerdem einen komfortablen Zugriff auf den Message–Header einer HTTP–Nachricht.

Libraray Interface

Alle Komponenten dieser Ebene sind von der Laufzeitumgebung entkoppelt. Sie interagieren ausschließlich über das bereitgestellte Interface des Runtime Abstraction Layers mit dem Client oder der Plattform. Sie stellt High–Level–Zugriffe auf die HTTP–Nachrichten zur Verfügung. In ihr sind folgende wichtige Komponenten zu finden:

- RWS_Router
- RWS_RequestEntity
- RWS_ResponseEntity

RWS_Router Diese Komponente bildet das Herz des *URI zu Controller mappings*. Er verwaltet alle dem System bekannt gemachten Controller und die für sie regsitrierten URIs. Seine Aufgabe ist unter anderem, anhand eines RWS_HttpRequest–Objektes den entsprechenden Controller zu ermitteln und die jeweilige HTTP–Funktion des Controllers aufzurufen. Dabei liegt es in seiner Verantwortung, bei nicht zustellbaren Requests, REST–konforme HTTP–Responses für den Client zu erstellen. Ein Beispiel ist hier der Zugriff auf eine nicht implementierte Funktion. Dieser Request muss vom RWS_Router automatisch mit „405 METHOD NOT ALLOWED“ und einem gesetzten Allow–Header im HTTP–Response beantwortet werden.

RWS_RequestEntity Die RequestEntity abstrahiert die einfachen Zugriffe des RWS_HttpRequests und ist so in der Lage, unsichtbar für den Entwickler, Transformationen des Message–Bodies, wie Entschlüsselung oder Dekompression, durchzuführen. Der Entwickler wird durch die RequestEntity–Klassen von der Implementierung von (RESTful) HTTP–spezifischen Mechanismen der Übertragung des Message–Bodies, entlastet.

RWS_ResponseEntity Analog zu der RWS_RequestEntity werden die Schreibzugriffe auf die Klasse RWS_HttpResponse um Funktionalitäten wie :

- Output–Buffering
- HTTP–Entity Kompression
- automatische Content-Length–Berechnung

- Sicherstellung des HTTP Aufbaus (send Head before Data)

ergänzt.

Application Layer

Der Application Layer stellt den eigentlichen Web Service dar. Er hat die Aufgabe, die eigentliche Logik für einen Request bereitzustellen. Eine Web Service ist in dieser Ebene soweit von der Laufzeitumgebung entkoppelt, dass sie bei einer Portierung auf eine andere Plattform lediglich für sie übersetzt und gegen die dort laufende *librws*gelinkt werden muss.

7.3 Implementation

Die Architektur der *Restful Web Service* Bibliothek ist darauf ausgelegt, für verschiedene Plattformen realisiert werden zu können. Aus Zeitgründen wurde der plattformspezifische Teil jedoch nur für eine mögliche Laufzeitumgebung implementiert. Dabei handelt es sich um, in Projekten der Firma Taskit vorrangig eingesetzten, Kombinationen aus:

- Linux als Betriebsystem (embedded Linux, Debian etch/squeeze),
- lighttpd als Embedded Webserver,
- und fastcgi als Bindeglied zwischen Webserver und Web Service.

Da sich die Umsetzung des systemspezifischen Teils hier auf die FastCGI Schnittstelle stützt, sind die damit entwickelten Web Services, auch unter jedem anderen Webserver mit FCGI Schnittstelle nutzbar.

7.3.1 Konventionen

Die librws verfolgt einen Objekt orientierten Ansatz in der Implementierung. Die Module der librws können als Klassen angesehen werden. Eine Klasse spiegelt sich in einer .c und einer .h Datei mit dem jeweiligen Klassennamen wieder. Zum Beispiel für die Kalsse *RWS_MyController* gibt es die Dateien *RWS_MyController.c* und *RWS_MyController.h* . Klassennamen beginnen immer mit dem Prefix *RWS_* und anschließend mit einem Großbuchstaben des eigentlichen Klassennamens. Die Methoden einer Klasse beginnen immer mit dem klein geschriebenen prefix *rws_*.

7.3.2 Die Main-Methode

Um eine Web Service mit der librws zu starten, muss die Datei RWS.h eingebunden sein. Bevor die librws verwendet werden kann, muss sie einmalig mit dem Aufruf der Funktion *rws_init* initialisiert werden. Als Parameter können der Funktion die Aufrufparameter der Kommandozeile übergeben werden (argc und argv). Die librws kann Konfigurationsparameter auch daraus übernehmen. Die Konfiguration eines Web Services mit der librws ist in Abschnitt 7.4.3 auf Seite 58 beschrieben.

Nach einer erfolgreichen Initialisierung (Returncode 0), kann der Web Service mit dem Aufruf der Funktion *rws_run* gestartet werden. Die Funktion *rws_run* wird erst zurückkehren, wenn der Web Service beendet wird. Das Beenden des Web Service ist momentan nur durch Senden eines SIGINT, SIGTERM oder SIGUSR1 möglich. Dabei ist zu beachten, dass SIGINT und SIGTERM einen *graceful shutdown* auslösen, d.h. es wird auf die Beendigung derzeit laufender Bearbeitungen von HTTP–Request gewartet. SIGUSR1 dagegen beendet alle laufenden Prozesse sofort. Achtung! Es werden in diesem Fall auch keine Deinitialisierungen mehr durchgeführt (Freigabe von Ressourcen, wiederherstellen von Signalmasken etc.).

Nach diesen zwei Schritten ist der Web Service bereits lauffähig, jedoch ohne Funktion. Er wird auf jede Anfrage mit dem Response Code *404 NOT FOUND* antworten, da er keinerlei Ressourcen verwaltet. Es muss nun eine Controller-Klasse implementiert und für eine bestimmte URI eingetragen werden. Die Implementierung einer solchen Controller-Klasse wird im nächsten Abschnitt auf Seite 51 erläutert. Ein Controller kann über die Funktion *rws_registerController* an eine URI gebunden werden. Das Binden einer URI an einen Controller sollte vor dem Starten des Web Services erfolgen. In jedem Fall ist darauf zu achten, dass die Initialisierung des librws durch *rws_init* bereits erfolgreich durchgeführt worden ist.

Es ist prinzipiell möglich, über die Funktionen *rws_registerController*
und *rws_unRegisterController* auch zur Laufzeit des Web Services, also nach dem Start durch *rws_run*, Controller für URI's an– und abzumelden. Beide Funktionen sind *thread safe* implementiert. Bei dem Versuch einen Controller für eine URI zu registrieren, die bereits an einen anderen Controller gebunden ist, wird die URI an den neuen Controller gebunden und eine Referenz auf den vorherigen Controller zurückgeliefert. Der Aufrufer ist in diesem Fall dafür verantwortlich, den Controller zu zerstören oder für eine spätere Nutzung zu verwalten.

7.3.3 Implementierung eines Controller

Für die Implementierung eines Controllers empfiehlt sich das Grundgerüst aus Listing 7.1. Wichtig ist in der Definition eines neuen Controllers, dass die neue *Controller–*

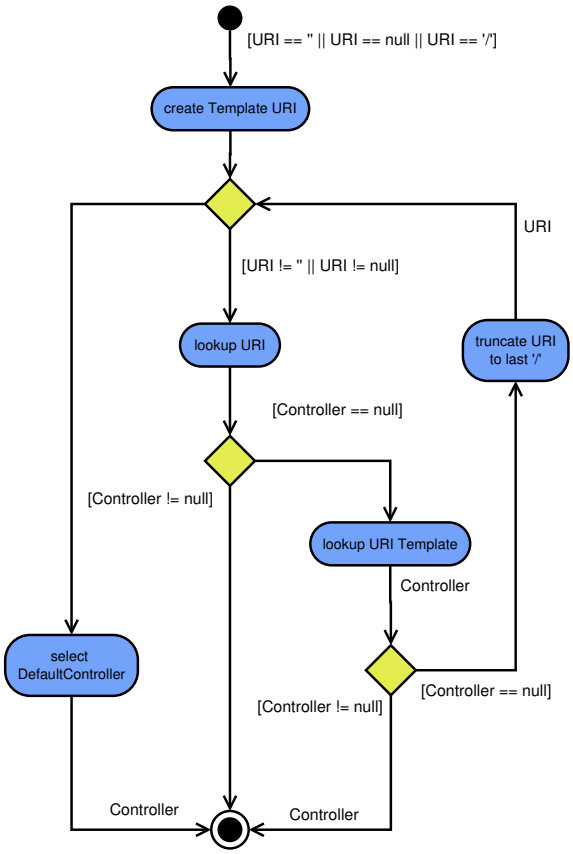

Abbildung 7.3: Das Diagramm zeigt den Vorgang des Auflösens einer URI zu einem Controller.

Struktur als erstes eine Struktur vom Typ *RWS_Controller* hat. Die Position ist sehr wichtig, da hierüber eine Art Vererbung realisiert wird. Das Ziel ist, jeden beliebigen Controller auf den Basis-Controller *RWS_Controller* casten zu können. Nach der Deklaration des Controllers können beliebige, für den jeweiligen Controller notwendige, Deklarationen erfolgen.

Ein Controller verwaltet alle Operationen auf URIs für die er registriert wurde. Das sind:

- GET
- PUT
- POST
- DELETE

```
typedef struct MyTestController {
  RWS_Controller _;
  int myVar;
  /* ... */
}
```

Listing 7.1: Definition einer neuen Controller Klasse

- HEAD
- OPTIONS
- CONNECT
- TRACE

. Ein Controller muss nicht jede HTTP–Methode für eine URI implementieren. Um eine Funtkion für eine HTTP–Methode zu registrieren wird die Funktion als Callback–Funktion der jeweiligen HTTP–Methode zugewiesen. Dies kann direkt über das Controller–Objekt erfolgen. Einfacher ist die Verwendung des Makros *RWS_CONTROLLER_REGISTER_HTTP_METHODES*. Dem Makro wird ein Pointer auf das Controller–Objekt, gefolgt von den Callback–Funktionen für die HTTP–Methoden, übergeben. Die Reihenfolge richtet sich nach der Definition in *RWS_Controller.h* (obige Aufzählung). Für nicht benötigte Funktionen wird *NULL* angegeben. Eine Besonderheit stellt die OPTIONS–Funktion dar. Ein *NULL*–Pointer für diese Funktion wird durch eine Standard–Funktion ersetzt. Diese Standard–Funktion erzeugt automatisch einen *RWS_HTTPResponse*, der einen Allow–Header mit den verfügbaren HTTP–Methoden des Controllers enthält. Ist dies ausdrücklich nicht gewünscht, muss sie nach der Initialisierung mit dem Makro *RWS_CONTROLLER_REGISTER_HTTP_METHODES* nochmals mit einem *NULL* überschrieben werden. Das Listing 7.2 zeigt wie Callback–Funktionen manuell im Controller registriert werden können. Zu beachten sind hier die Casts. Es können nun neue Controller–Instanzen erzeugt und für eine URI registriert werden.

```
MyController* new MyController(){
  MyController *lController = NULL;
  /* malloc initialise lController ...*/

  ( (RWS_Controller*) lController )->httpGet = &myController_get;
  ( (RWS_Controller*) lController )->httpPut = &myController_put;
  ( (RWS_Controller*) lController )->httpPost = NULL;
  ( (RWS_Controller*) lController )->httpDelete = NULL;
  ( (RWS_Controller*) lController )->httpHead = NULL;
  ( (RWS_Controller*) lController )->httpOptions = NULL; /* no automatic
     allow Response for OPTIONS-Requests */
  ( (RWS_Controller*) lController )->httpConnect = NULL;
  ( (RWS_Controller*) lController )->httpTrace = NULL;

  /*...*/
}
```

Listing 7.2: manuelle Registrierung von HTTP–Callback–Funktionen

Bei einem eingehenden Request für eine URI wird zunächst überprüft, ob ein Controller für sie registriert ist und anschließend ob der Controller über eine passende Callback–Funktion für die HTTP-Methoden verfügt. Die HTTP–Callback–Funktion wird mit einer Referenz auf den Controller und das Request–Objekt aufgerufen. An dieser Stelle setzt die eigentliche Bearbeitung des Requests ein. Ein *RWS_HTTPResponse*–Objekt dient zur Ausgabe der *HTTP–Header* und der *HTTP-Message*. Von der HTTP–Callback–Funktion wird ein solches *RWS_HTTPResponse*–Objekt erwartet. Gibt eine HTTP–Callback–Funktion stattdessen *NULL* zurück, wird von einem internen Fehler ausgegangen und die Bibliothek antwortet dem Client mit dem HTTP–Status *500 INTERNAL SERVERERROR*.

7.3.4 Verarbeitung eines Requestes

Die Grundlage für die Bearbeitung eines *HTTP-Request* sind die beiden Objekte

RWS_HTTPResponse und *RWS_HTTPRequest*.

Sie abstrahieren die zugrunde liegende Laufzeitumgebung durch eigene Methoden für den Zugriff auf HTTP-Header und zum Senden und Empfangen. Die Abbildung 7.4 zeigt den Ablauf der Request–Response–Kette als Sequenz Diagramm.

Beide Objecte sind über entsprechende Member-Variablen miteinander verknüpft. Zu einem Request gehört immer ein Response und umgekehrt. Daraus resultiert die Einschränkung, ein *RWS_HTTPResponse*-Objekt nur mit einem *RWS_HTTPRequest* erzeugen zu können. Wie am Ende des vorherigen Abschnittes schon erwähnt ist, muss jede HTTP–Callback–Funtkion eines Controllers ein *RWS_HTTPResponse*-Objekt zurück liefern. Damit sieht der minimale Aufbau einer HTTP–Callback–Funktion wie in Listing 7.3 aus.

Da ein *RWS_HTTPResponse*-Objekt sowohl für erfolgreiche Antworten, wie auch für Fehlermeldungen benötigt wird, empfiehlt es sich, gleich zu Beginn der Request-Verarbeitung eines zu erzeugen. Sollte der Konstruktoraufruf *NULL* zurück liefern, kann die Bearbeitung mit der Rückgabe von *NULL* beendet werden. Der Client erhält dadurch einen Response mit dem Status-Code *500 INTERNAL SERVERERROR*.

Das *RWS_HTTPRequest*–Objekt ermöglicht über die Funktionen
rws_HttpResponse_setMetaTag und *rws_HttpResponse_getMetaTag*

Zugriff auf den HTTP–Request–Header. Über die Funktion *rws_HttpRequest_read* können vom Client übermittelte HTTP-Entities gelesen werden. Sie verhält sich wie die aus der Standard-C-Bibliothek bekannte *read*–Funktion.

Nach der Auswertung des Request können nun der Status-Code und die entsprechen-

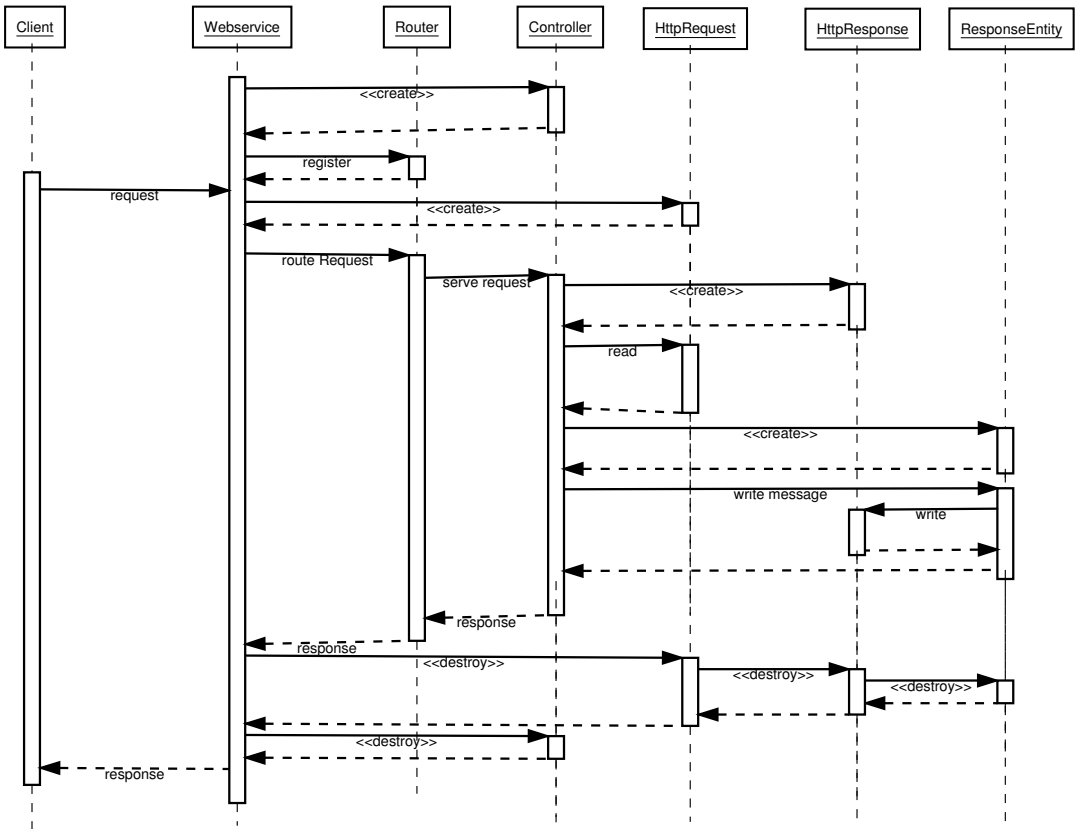

Abbildung 7.4: Das Sequenz–Diagramm zeigt die Abarbeitung eines HTTP–Requests in der *librws*.

den HTTP-Response-Header gesetzt werden. Der Status-Code wird über die Eigenschaft *responseCode* des *RWS_HTTPResponse*-Objekts gesetzt. Hierfür stehen Pre-Compiler-Defines in der Form *RWS_HTTP_*Statusmessage zur Verfügung. Über die Funktionen *rws_HttpResponse_setMetaTag* und *rws_HttpResponse_getMetaTag* lassen sich die HTTP–Response–Header setzen und erfragen. Die Funktionen *rws_HttpResponse_write* und *rws_HttpResponse_printf* dienen zur Ausgabe des HTTP–Response. Zu beachten ist bei der Verwendung der write–Funktionen, dass zuvor alle HTTP–Header im Response–Objekt gesetzt wurden, da der erste Aufruf einer write–Funktionen des Response–Objekts das Schreiben des HTTP–Headers auslöst.

Komfortabler ist die Verarbeitung unter Verwendung von Request- bzw. ResponseEntities. Diese Objekte kapseln den Lese- und Schreibzugriff auf das zugrunde liegende Request– und Response–Objekt. Dadurch wird eine Vorverarbeitung des Request und Response möglich. So zum Beispiel:

- De-/und Kompression von HTTP–Entities
- Ver- und Entschlüsselung von HTTP–Entities

```
1
RWS_HttpResponse* http_method_callback( RWS_Controller * const pController,
    RWS_HttpReuqest * const pRequest ){
3   RWS_Response *IResponse = NULL;

5   /* create new RWS_HttpResponse object */
    IResponse = new_RWS_Response( pReuqest, NULL );
7   if( IResponse == NULL )
      return NULL;
9   /*
      IResponse->responseCode = RWS_HTTP_OK;
11    ...
    */
13
    return IResponse;
15 }
```

Listing 7.3: Controller C–Datei

- gepufferte Ein-/Ausgabe

Über den Ansatz der Vererbung, wie in Abschnitt 7.3.1 auf Seite 50 beschrieben, können aus den beiden Basis–HTTP–Entities *RWS_RequestEntity* und *RWS_ResponseEntity* weitere Abstraktionen für unterschiedliche Aufgaben abgeleitet werden. *RWS_ErrorDocument* ist eine solche von *RWS_ResponseEntity* abgeleitete HTTP–Entity. Sie erzeugt für einen HTTP–Response–Code und einer optionalen Fehlermeldung eine zu den HTTP–Request–Headern passende Repräsentation.
Ohne weitere Konfiguration wird eine *RWS_ResponseEntity* die Ausgabe in einen internen Puffer zwischenspeichern. Ist dieses Verhalten nicht gewollt, muss es explizit vor dem ersten Schreibvorgang mit dem Makro

RWS_RESPONSE_OUTPUT_BUFFER_OFF()

deaktiviert werden. Ähnlich verhält es sich mit der Kompression. Es wird versucht, anhand des Request–Headers *CONTENT-ENCONDIG*, ein geeignetes Kompressionsformat zu wählen.

7.4 Installation und Konfiguration

7.4.1 Voraussetzungen

Für die Nutzung der librws müssen folgende Anforderungen erfüllt sein:

- ISO C99 (variable Makros)
- libc (standard c libraray, malloc/free)
- libtrie (liegt bei)

- libz
- pkg-config
- make (gnu)

Je nach Zielplattform kommen folgende Abhängigkeiten hinzu:

für FCGI:

- pthread
- libfcgi (FCGI-SDK oder fcgi–Packet der Distribution)

für mongoose:

- pthread
- mongoose (bereits integriert)

7.4.2 Erstellen der librws

Die *librws* verwendet für den *Build*–Prozess *pkg–config*. Dadurch ist eine einfache Integration in das automatische Buildsystem des Embedded–Linux–Projektes für die Erstellung eines System–Images oder die optionale Bereitstellung über den Packetmanager gewährleistet. Über das *pkg-config*–Werzeug werden die benötigten Kompiler– und Linker– Flags, sowie Include–Verzeichnisse der Header-Dateien und Bibliothekspfade für die benötigten externen Bibliotheken des Zielsystems ermittelt. Sind alle Vorraussetzungen erfüllt wird die *librws* mit dem Aufruf von *make* erstellt. Das Verhalten während des make–Durchlaufes kann über verschiedene Parameter beeinflusst werden. Im Folgenden werden die *librws* spezifischen Parameter aufgezählt und die Standardeinstellungen angegeben.

RWS_SERVER Gibt an, für welche Ziel–Laufzeitumgebung die Bibliothek erstellt wird. Es stehen momentan nur RWS_FCGI und RWS_MONGOOSE zur Verfügung. In der Standardkonfiguration wird mit der Option RWS_SERVER=RWS_FCGI kompiliert. Die Unterstützung für Mongoose ist allerdings nur experimentell und nicht ausführlich getestet.

DESTDIR Der Parameter verhält sich wie der aus GNU–Make bekannte und gibt den Installationspfad an. Er wird im Makefile nicht gesetzt und ist somit leer.

PREFIX Auch der Prefix-Parameter ist aus den GNU–Make–Guidelines übernommen und verhält sich dementsprechend. Über ihn kann ein Verzeichnispfad für die spätere Installation im Installations–Verzeichnis (DESTDIR) angegeben werden. Die *PREFIX*–Variable ist mit */usr/local* vorbelegt.

Das Makefile bietet folgende Targets an:

all
: startet den *Build*–Prozess

install
: installiert die zur Bibliothek gehörenden Dateien an den entsprechenden Stellen im System. Dazu gehören Header–Dateien (*rws/*), statische und dynamische Bibliotheken und PKG–Config–Datei (librws.pc)

Die manuelle Installation erfolgt durch Ausführen der folgenden 2 Kommandos:

```
$>make
...
#>make install
```

Listing 7.4: erstellen und installieren der librws

Je nachdem welche Werte für *PREFIX* und *DESTDIR* gesetzt sind, werden für den Aufruf von *make install* root–Rechte benötigt.

7.4.3 Konfiguration eines *librws*–Web Service

librws–Web Services können über das Environment konfiguriert werden. Dafür werden einfache Umgebungsvariablen in der Form Key=Value festgelegt. Folgende Variablen können gesetzt werden:

RWS_ENV_SERVICE_NAME
: legt den Namen fest, mit dem der Web Service sich in Log–Files identifizieren lässt (Standard: Inhalt von *PWD* sonst *RWebService*).

RWS_ENV_SERVICE_THREADS
: legt die Anzahl der verwendeten Service–Threads für die Bearbeitung von HTTP–Request fest (Standardwert: 6).

RWS_ENV_LOG_FILE
: dient zum Festlegen einer Logdatei. (Standard: stderr)

RWS_ENV_LOG_LEVEL
: hier kann eine Kombination der 4 möglichen Log–Prioritäten angegeben werden DEBUG, INFO, WARNING, ERROR (Standard: alle)

RWS_ENV_MEMORY_MAX
: Die *librws*setzt ein Memory–Hard–limit für den eigenen Prozess. Die Angabe des maximalen Arbeitsspeichers wird in Byte angegeben.

8 Verifizierung von Web Services (Debug&Test)

Embedded Web Services werden in den meisten Fällen für Steuer– und Kontrollaufgaben zuständig sein. Die Anbindung über ein Web Service an das Internet erfordert deshalb besondere Sorgfalt. Waren die Steuersysteme ursprünglich nur über dedizierte Computersysteme zugänglich, ist über einen RESTful Web Service der Zugriff von jedem internetfähigen Gerät möglich. Somit ergeben sich zusätzliche Fehlerquellen, die zu untersuchen sind:

1. Fehlbedienung durch den Nutzer
2. unterschiedliche Implementation / Interpretation von Standards (HTTP Deflate: zlib oder deflate)
3. fehlerhaft genutzte Middleware (Webserver , Bibliotheken)
4. Fehler in der Implementierung des Web Service (Programmierfehler)
5. unterschätzter Ressourcenverbrauch (intensivere Nutzung des Service als vorhergesehen, zu große Datenmengen)
6. unzureichende Sicherung (Authentifizierung/ Autorisierung)

Ein schwieriges Problem stellen die Unterschiede in den Implementierungen des HTTP–Protokoll dar. Ein Beispiel dafür ist der Kompressions–Algorithmus für *Content-Encoding* mit *deflate*, der im [Fielding et al.(1999)Fielding, Gettys, Mogul, Frystyk, Masinter, Leach, and Berners-Lee] des HTTP angegeben wird. Hier wird für das Schlüsselwort *deflate* sowohl der *zlib* ([Deutsch and Gailly(1996)]) als auch der *raw-deflate* ([Mealy(1971)]) Algorithmus angegeben. Probleme entstehen hier, wenn HTTP–Client oder –Server den *raw-deflate*, anstatt des *deflate (zlib)* verwenden. Moderne Browser unterstützen häufig beide Versionen, jedoch funktioniert es nicht mit dem Microsoft Explorer sowie auf Programmen, welche auf der Klasse *System.IO.Compression.DeflateStream* des .Net-Framework basieren. Diese Formen der unterschiedlichen Implementierung von Protokolldetails gilt es, bei der Entwicklung und besonders während der Testphase zu berücksichtigen. Eventuelle Inkompatibilitäten durch ein zu enges Testfeld von Client–Anwendungen können sonst unentdeckt bleiben oder zu langwierigen unnötigen Fehlersuchen am Web Service führen.

Eine typische Gruppe von Fehlern, sind die Programmierfehler. Besonders ein in *C* umgesetzter Web Service, ist bei unzureichender oder nicht vorhandener Parameterprüfung auf sinnvolle Grenzwerte, eine enorme Bedrohung für die Stabilität und Sicherheit des gesamten Embedded–Systems. Die Palette der Risiken reicht hier von einem Systemausfall bis hin zur Übernahme der Steuerung durch unautorisierte Dritte. Eine weitere mögliche Fehlerquelle kann ein unerwartet hoher Ressourcenverbrauch darstellen. Dafür kommen folgende Möglichkeiten in Betracht:

- Ein schlechter oder verschwenderischer Umgang mit Systemressourcen in der Implementierung.
- Eine falsche oder unzureichende Kalkulation der benötigten Systemressourcen für die Erstellung oder Verarbeitung von HTTP–Response und –Request.
- Ein intensivere Nutzung des Web Service als vorhergesehen.

Alle diese Punkte können dazu führen, dass das System seiner eigentlichen Aufgabe (Steuern, Messen, etc.) nicht mehr nachkommt oder ganz ausfällt.

Der letzte wichtige Schwerpunkt zum Test von Embedded Web Services, ist die Überprüfung der Zugangsberechtigung für kritische Ressourcen. Hierbei ist es wichtig, alle Ressourcen auszumachen, welche eine oder mehrere Methoden besitzen, die nur autorisierten Nutzern zur Verfügung stehen und diese gezielt zu testen. Dabei steht nicht die Sicherheit des verwendeten Autentifizierungsmechanismus im Vordergrund, sondern ob:

- geschützte Ressourcen ohne Autorisierung abrufbar sind (HTTP–GET, HTTP–HEAD),
- geschützte Ressourcen ohne Autorisierung veränderbar sind (HTTP–PUT),
- Ressourcen ohne Autorisierung hinzugefügt werden können (HTTP–POST),
- geschützte Ressourcen ohne Autorisierung gelöscht werden können (HTTP–DELET),
- die Zuordnung der Autorisierung zu den Methoden der Ressourcen eindeutig ist.

Der letzte Punkt bezieht sich auf eine mögliche fehlerhafte Implementierung der Zugriffsverwaltung. So kann ein authentifizierter Benutzer, welcher für einen Zugriff auf die Ressource */resources/a* die notwendige Autorisierung besitzt, gleiche aber unberechtigte Zugriffsrechte auf Ressource */resource/b* erhalten.

8.1 Vorbereitungen und Voraussetzungen

Bereits in der Entwurfsphase sollten Schnittstellen für das spätere Testen und Debuggen berücksichtigt werden. Solche Ansatzpunkte vereinfachen wesentlich die spätere Fehlersuche, da hier aus dem Programmkontext heraus Informationen gewonnen werden können. Informationen, die durch externe Werkzeuge zur Laufzeit ermittelt werden, sind oft schwieriger in den richtigen Zusammenhang zu setzen und benötigen so mehr Aufwand um zu einem gleichwertigen Ergebnis zu kommen.

8.1.1 Bedingte Kompilierung

Der *C*–Preprozessor ist ein mächtiges Werkzeug und eignet sich sehr gut, um einfach verschiedene Programmversionen erzeugen zu können. Mittels bedingter Kompilierung können Debug–Ausgaben oder *Asserts* nur für Testzwecke eingebunden werden. Listing 8.1 zeigt eine Debug–Ausgabe, die nur kompiliert wird, wenn die Prekompiler–Variable *DEBUG* gesetzt ist. Diese können beim Aufruf des Kompilers mit dem Schalter *-D* und in diesem Fall *-DDEBUG=1* übergeben werden.

```
1 #ifdef DEBUG
  if( ( 1 + 1 ) != 2 )
3 fprintf( stderr, "[%s:%d] 1 + 1 ist nicht mehr 2 !", __FILE__, __LINE__ );
  #endif
```

Listing 8.1: Bedingte Kompilierung wenn DEBUG gesetzt ist

8.1.2 Compiler–Flags

Für späteres Debuggen und Profiling muss der Compiler angewiesen werden, entsprechende Debug–Informationen während des Erstellens einzubinden. Folgende Schalter erweisen sich als nützlich:

-g*N*
: weist den Kompiler an, Debug–Informationen einzubinden. *N* kann eine Zahl zwischen 1-3 annehmen, wobei 3 als ausführlichstes Level auch Makro–Definitionen für den Debugger enthält

-pg
: diese Option aktiviert Profiling–Informationen, die beispielsweise von *gprof* ausgewertet werden können

-a
: erlaubt zusätzliche Profiling–Information über Ausführungshäufigkeit von Programmblöcken

-Wextra
: in der Debug–Phase können alle Warnungen des Kompilers wichtige Hinweise geben

-pedantic / **-pedantic-errors**
: führt zu weiteren Warnungen (bzw. Fehlern) bei nicht ISO-C konformen Praktiken

In größeren Projekten, in denen ein build–Werkzeug ähnlich gnu–Make verwendet wird, lohnt es sich, die Debug–Flags ähnlich der bedingten Kompilierung an die CFLAGS anzuhängen.

```
ifdef DEBUG
  CFLAGS+= -pg -g3 -Wextra -pedantic
endif
```

Listing 8.2: Wenn DEBUG gesetzt ist, wird die CFLAGS–Variable für Debug–Informationen zur Kompile– und Laufzeit ergänzt

8.2 Testen von Embedded Web Services

Nach Fertigstellung des Web Services gilt es zu verifizieren, ob er den notwendigen Anforderungen entspricht. Dafür können drei Testkategorien festgehalten werden:

- Funktionstest
- Performancetest / Belastungstest
- Stabilitätstest

8.2.1 Funktionstest

Grundlegend sollte die Funktion des Web Services nachgewiesen werden können. Auch im Embedded–Bereich kann hier auf die bewährte Methodik der Unittests zurückgegriffen werden. Während dieser Arbeit kam das Framework *CUnit* zum Einsatz. Die Vorgehensweise unter Verwendung von Unittests birgt den Vorteil, dass Teile des Web Services mit Werkzeugen untersucht werden können, die für die Zielplattform nicht zur Verfügung stehen. Dafür genügt es, die entsprechenden Komponenten des Web Services mit den zugehörigen Unittests separat auf einem Entwicklungssystem zu übersetzen und mit den entsprechenden Werkzeugen zu analysieren. Detaillierter wird das im Abschnitt 8.3 behandelt.

Es können nun für jede Ressource und deren Methoden Testfälle konstruiert werden. Um die Nutzung von HTTP zu erleichtern, bietet sich die *C*–Bibliothek libcurl an. Alternativ können die Unittests auch in jeder anderen beliebigen Programmiersprache implementiert werden. Für die Bibliothek *libcurl* existieren für viele Programmiersprachen entsprechende Library–Bindings, unter anderem für C/C++, Java, PHP, Python, Perl, Lua, TCL und .NET. Die Unittests können wahlweise auf dem Zielsystem oder auf einem separaten Entwicklungssystem implementiert und ausgeführt werden. Es empfiehlt sich, pro Ressource eine Testsuite anzulegen, welche jeweils für jede HTTP–Methode mindestens einen Testfall implementiert. Auch wenn die eine Ressource nicht alle HTTP–Methoden unterstützt, sollte geprüft werden, ob der Web Service in diesen Fällen mit einem entsprechenden Response–Code *405 METHODE NOT ALLOWED* antwortet.

8.2.2 Performancetest & Belastungstest

Für das Testen eines Web Services eignen sich die Kommandozeilenwerkzeuge *httperf* und *ab* (Apache Benchmarking Tool). Mit ihrer Hilfe ist es möglich, verschiedene HTTP–Requests zu erzeugen. Dabei kann angegeben werden, wie viel insgesamt und davon gleichzeitig an den Server geschickt werden sollen. Die statistischen Auswertungen beider Werkzeuge helfen, die Leistung des Systems einzuschätzen und machen deutlich, welche Ressourcen bzw. Methoden einer Ressource eine lange Antwortzeit nach sich ziehen.

8.2.3 Stabilitätstest

Die Stabilitätstest dienen dazu, die Grenzen des Systems aus zu ermitteln. Dafür wird der Web Service mit einer steigenden Anzahl an Request belastet, um herauszufinden, ab welcher Grenze er ausfällt. Dabei ist es wichtig, den Ausfall zu protokollieren. Einen Ausfall unter oder auf der festgelegten Belastungsobergrenze bedeutet einen Fehlschlag. Anhand von Log–, Debug– oder Profiling–Informationen lassen sich die Ursachen für den Ausfall ausmachen und so Maßnahmen gegen einen Ausfall durch Überlastung ergreifen. Diese Tests können mit einfachen Shell Scripts realisiert werden.

8.3 Debuggen von Embedded Web Services

Sind während des Testens Fehler aufgetaucht, gilt es deren Ursache aufzuspüren und zu beseitigen. Leider sind in den meisten Fällen die Fehler nicht trivial. Das bedeutet, sie können nicht durch die „Methode des scharfen Hinsehens“ gefunden werden. Es gibt eine Vielzahl von Tools zum Debuggen von Software; kommerzielle wie auch frei verfügbare. Die Möglichkeit, moderne Linux–Distributionen auf dem Zielsystem zu verwenden, spricht für frei verfügbare Software. Es existieren verschiedene Methoden, um Laufzeitfehlern zu finden. Sie lassen sich in folgende Gruppen aufteilen:

- Logging
- Laufzeitdebugger
- Profiler

Sie unterscheiden sich in ihrer Anwendung, Aussagekraft und Zuverlässigkeit. So zum Beispiel ist das sogenannte „printf“–Debugging oder auch *loggen* von Zuständen im Programmverlauf in Singel–Thread Anwendungen hilfreich und in den meisten Fällen auch ausreichend genau. In Multithreaded–Anwendungen kann es zu falschen Schlussfolgerungen führen.

Im Folgenden werden diese Methoden näher erläutert. Dabei wird auf Besonderheiten für die Anwendung im Embedded–Bereich eingegangen.

8.3.1 Logging

Das Loggen oder auch bekannt als *printf*–Debugging ist ein einfaches und effektives Werkzeug für die Fehlersuche. Die Ausgaben können durch das Betriebsystem gepuffert werden und müssen nicht mit dem zeitlichen Ablauf des Programms übereinstimmen. In Multithreaded–Anwendungen kann zusätzlich die Reihenfolge der Ausgaben variieren, wodurch es zu irreführenden Annahmen des Programmablaufes kommen kann. Durch diese Art von Logging verlängert sich die Programmlaufzeit um die Ausgaben. Hierbei ist zu bedenken, dass sich hinter der Gruppe der printf-Funktionen Systemaufrufe verbergen. Diese führen zu einer unvorhersehbaren Ausführungszeit des Programms. So kann beispielsweise der Schreibvorgang für einen Logeintrag auf einer SD–Karte die Auswertungsroutine für ein eingehendes Signal derart verzögern, dass das Signal nicht korrekt eingelesen wird. Bei Echtzeitanwendungen ist somit Vorsicht geboten.

8.3.2 Laufzeitdebugger

Eine der effektivsten Methoden zur systematischen Fehlersuche ist ein Laufzeitdebugger. Mit seiner Hilfe können Programme während der Laufzeit angehalten und untersucht werden. Der gnu–Debugger ist ein bekannter Vertreter. Nützliche Features des gdb sind:

- Bedingte Breakpoints
- einklinken in laufende Prozesse
- direkter Speicherzugriff
- remote debugging

8.3.3 Profiling

Das *Profiling* eines Web Services ermöglicht das Auffinden von Fehlern im Laufzeitverhalten, die mit anderen Methoden nur sehr schwer oder gar nicht auszumachen sind. Ein Vertreter dieser Fehler ist der *Memory-Leak* oder auch das *Speicherloch* genannt. Dieser Fehler führt, insbesondere bei Anwendungen, welche über längere Zeiträume laufen, zu einem extrem hohen Speicherverbrauch. Alloziert der Web Service Speicher auf dem *Heap* und gibt diesen aber nicht wieder frei, wird so der verfügbare Arbeitsspeicher zu immer größeren Teilen von einem Prozess blockiert. Ein Beispiel ist ein *Request*–Objekt, welches bei jeder Anfrage eines Clients dynamisch auf dem *Heap*

erzeugt wird. Dabei ist die Größe der Datenstruktur oder damit verbundenen auch dynamisch allozierten Objekten nicht ausschlaggebend. Entscheidend ist die sich häufende Reservierung. An diesem Punkt setzen Profiler an. Sie protokollieren den Programmablauf auf bestimmte Schwerpunkte hin. So z.B.:

- alozierte Speicherobjekte
- Anzahl Funktionsaufrufe

Mit Hilfe dieser Analysen kann anschließend gezielt das Laufzeitverhalten im Hinblick auf Speicherverbrauch und Ausführungsgeschwindigkeit optimiert werden.

mtrace

Das Tool *mtrace* ist in der gnu compiler toolchain enthalten. Mit seiner Hilfe werden während der Laufzeit des Programmes die Systemrufe zu malloc, calloc und realloc sowie die zugehörigen Aufrufe von *free*, geloggt. Es wird so ermöglicht festzustellen, ob und wieviel Speicher alloziert und nicht wieder freigegeben wird. Um mtrace zu aktivieren, kann es zum einen im Quellcode direkt eingebunden werden, zum anderen kann durch zusätzliches Linken gegen die Bibliothek -libmtrace ein Programm mit mtrace–Unterstützung erzeugt werden. Konfiguriert wird mtrace über die Umgebungs-

```
1  #include <mtrace.h>
   #include <stdlib.h>
3  #include <stdio.h>

5  int main( int pArgc, char *pArgv[] ){
     char *lString = NULL;
7    /* init mcheck NULL or callback function pointer */
     mcheck( NULL );
9    mtrace();

11   /* ... */

13   /* explicit test allocated memory block */
     mprobe( lString );
15 }
```

Listing 8.3: Einfaches C–Programm zur Demonstration von mtrace

variablen.

valgrind

Valgrind bietet verschiedene Tests, mit denen das Laufzeitverhalten eines Programms untersucht werden kann. Allerdings enthält Valgrind in der aktuellen Version 3.8.0 keinen Support für den auf dem PicosG20 eingesetzten Mikroprozessor mit arm5v Architektur.

Als Workaround bleibt hier die Möglichkeit, unter Verwendung von Unittests Teile des Systems auf einer unterstützten Plattform zu übersetzen und mit valgrind zu testen. Um einen Web Service mit valgrind in einer FCGI–Umgebung testen zu können, kann ein einfaches Shell Skript verwendet werden. Das Script in Listing 8.4 zeigt ein Beispiel zum Start eines Web Services mit valgrind.

```
#!/bin/bash

FCGI_PROG="/home/devel/wetterst/wetter.fcgi"
PARAM=

VAL_ENABLE=0
################################################
VALGRIND="/usr/local/bin/valgrind"
VALGRIND_LOG="/tmp/${0##*/}.log"
VALGRIND_TOOL="memcheck"
VALGRIND_OPTIONS="-v --tool=$VALGRIND_TOOL --leak-check=full --log-
    file=$VALGRIND_LOG"

if [ $VAL_ENABLE ]; then
        echo "$VALGRIND $VALGRIND_OPTIONS $FCGI_PROG $PARAM"
        $VALGRIND $VALGRIND_OPTIONS $FCGI_PROG $PARAM
else
        echo "exec $FCGI_PROG $PARAM <&0" > &2
fi
```

Listing 8.4: Mit Hilfe dieses Scripts können FCGI–Programme mittels valgrind überprüft werden.

9 Zusammenfassung und Ausblick

Das HTTP und damit verbundene Techniken wie z.B. HTML, URI und CSS unterliegen einer stetigen evolutionären Weiterentwicklung. Nach 24 Jahren ist es bis heute möglich sowohl Server als auch Clients älterer Generationen in das WWW einzubinden. Die Abwärtskompatibilität, die weite Verbreitung und eine nicht abzusehende Ablösung des HTTP genügen den Ansprüchen an die Langlebigkeit von Embedded Systemen.

Der Vergleich der beiden in der Enterprise–Welt geläufigen Software–Architektur–Stile SOA und ROA hat gezeigt, dass HTTP in Verbindung mit RESTful Web Services, als Vertreter einer Ressourcen–orientierten Architektur, den besseren softwaretechnischen Ansatz für Embedded Web Services bietet. Durch die wachsende Leistung von eingebetteten Systemen im Zusammenspiel mit der rasanten Weiterentwicklung mobiler Endgeräte wird RESTful HTTP in naher Zukunft auch in professionellen und industriellen Einsatzgebieten an Bedeutung gewinnen. Sowohl in der Mensch–Maschine– als auch in der Maschine–Maschine–Kommunikation werden die in der Enterprise–Welt erprobten Techniken einer Ressourcen–orientierten Architektur in Verbindung mit dem HTTP und REST–konformen Schnittstellen–Design die softwaretechnischen Grundlagen bilden.

Als praktische Ergebnisse dieser Arbeit entstand eine in *C* geschriebene Bibliothek (li$\tilde{\text{b}}$rws). Diese wurde im praktischen Einsatz mittels eines Wetter–Web–Service und eines ModbusIO–Web Service getestet. Parallel entstand in Zusammenarbeit mit der Firma *taskit* und dem *Elektor*–Verlag ein Artikel zum Thema RESTful Web Services für *Embbeded Systems* unter dem Titel „WEB–ArBYTER“. Dieser wurde in der Sonderausgabe „Mikrocontroller 7 – Embedded Systems in der Praxis“ Januar 2013 veröffentlicht.

Literaturverzeichnis

[W3C()] W3c device apis working group. URL http://www.w3.org/2009/dap/.

[Berners-Lee(1990)] T. Berners-Lee. Hypertext and cern. W3C, März 1990. URL http://www.w3.org/Administration/HTandCERN.txt.

[Berners-Lee et al.(1994)Berners-Lee, Masinter, and McCahill] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). RFC 1738 (Proposed Standard), Dec. 1994. URL http://www.ietf.org/rfc/rfc1738.txt. Obsoleted by RFCs 4248, 4266, updated by RFCs 1808, 2368, 2396, 3986, 6196, 6270.

[Berners-Lee et al.(1998)Berners-Lee, Fielding, and Masinter] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396 (Draft Standard), Aug. 1998. URL http://www.ietf.org/rfc/rfc2396.txt. Obsoleted by RFC 3986, updated by RFC 2732.

[David Booth(2004)] W. F. M. F. L. o. A. E. N. u. O. . I. M. C. u. M. . S. A. C. F. u. M. . I. D. O. u. M. . B. S. David Booth, W3C Fellow / Hewlett-Packard Hugo Haas. Web services architecture. W3C, February 2004. URL http://www.w3.org/TR/ws-arch/.

[Deutsch and Gailly(1996)] P. Deutsch and J.-L. Gailly. ZLIB Compressed Data Format Specification version 3.3. RFC 1950 (Informational), May 1996. URL http://www.ietf.org/rfc/rfc1950.txt.

[Duerst and Suignard(2005)] M. Duerst and M. Suignard. Internationalized Resource Identifiers (IRIs). RFC 3987 (Proposed Standard), Jan. 2005. URL http://www.ietf.org/rfc/rfc3987.txt.

[Fette and Melnikov(2011)] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), Dec. 2011. URL http://www.ietf.org/rfc/rfc6455.txt.

[Fielding et al.(1999)Fielding, Gettys, Mogul, Frystyk, Masinter, Leach, and Berners-Lee] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. URL http://www.ietf.org/rfc/rfc2616.txt. Updated by RFCs 2817, 5785, 6266, 6585.

[Fielding(2000)] R. T. Fielding. *Architectural Styles and the Design of Netwaork-based Software Architektures*. PhD thesis, University of California, Irvine, 2000.

[Franks et al.(1999)Franks, Hallam-Baker, Hostetler, Lawrence, Leach, Luotonen, and Stewart] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard), June 1999. URL http://www.ietf.org/rfc/rfc2617.txt.

[Freed and Klensin(2005)] N. Freed and J. Klensin. Media Type Specifications and Registration Procedures. RFC 4288 (Best Current Practice), Dec. 2005. URL http://www.ietf.org/rfc/rfc4288.txt.

[Hickson(2009)] I. Hickson. The web sockets api, Dezember 2009. URL http://www.w3.org/TR/2009/WD-websockets-20091222/.

[Hickson(2012)] I. Hickson. Server-sent events, April 2012. URL http://www.w3.org/TR/eventsource/.

[Horn(2008)] T. Horn. Soa (service oriented architecture). Torsten Horn, 2008. URL www.torsten-horn.de/techdocs/soa.htm.

[Leonard Richardson(2007)] S. R. Leonard Richardson. *Web Services mit REST*. O'REILLY, 2007.

[Mealy(1971)] G. Mealy. Data computers-data descriptions and access language. RFC 195, July 1971. URL http://www.ietf.org/rfc/rfc195.txt.

Anhang A: HTTP–Response–Codes

Tabelle A.1: **HTTP-Status-Codes und ihre Bedeutung**

Statuscode	Bezeichnung	Erläuterung
100	Continue	Häufig bei großen Anfragen an den Servern der Fall. Die Anfrage wurde noch nicht zurckgewiesen und der Client kann fortfahren.
101	Switching Protocols	Wird verwendet, wenn der Server mit dem Wechsel zu einem anderen Protokoll einverstanden ist.
102	Processing	Wird verwendet, um Timeouts zu vermeiden
200	OK	Die Anfrage wurde erfolgreich bearbeitet und das Ergebnis der Anfrage wird in der Antwort übertragen.
201	Created	Die Anfrage wurde erfolgreich bearbeitet. Die angeforderte Ressource wurde vor dem Senden der Antwort erstellt.
202	Accepted	Die Anfrage wurde akzeptiert, wird aber zu einem späteren Zeitpunkt ausgeführt.
203	Non-Authoritative Information	Die Anfrage wurde bearbeitet, das Ergebnis ist aber nicht unbedingt vollständig und aktuell.
204	No Content	Die Anfrage wurde erfolgreich durchgeführt, die Antwort enthält jedoch keine Daten.
205	Reset Content	Die Anfrage wurde erfolgreich durchgeführt, der Client soll das Dokument neu aufbauen und Formulareingaben zurücksetzen.
206	Partial Content	Der angeforderte Teil eines Downloads wurde erfolgreich übertragen.
207	Multi-Status	Die Antwort enthält ein XML-Dokument, das mehrere Statuscodes zu unabhängig voneinander durchgeführten Operationen enthält.
300	Multiple Choice	Die angeforderte Ressource steht in verschiedenen Arten zur Verfügung.
301	Moved Permanently	Die angeforderte Ressource steht ab sofort unter der im „Location“-Header-Feld angegebenen Adresse bereit. Die alte Adresse ist nicht mehr gültig.

Continued on next page

Tabelle A.1 – continued from previous page

Statuscode	Bezeichnung	Erläuterung
302	Found	Die angeforderte Ressource steht vorübergehend unter der im „Location“-Header-Feld angegebenen Adresse bereit (in HTTP/1.0 „Moved Temporarily“). Die alte Adresse bleibt gültig. Wird in HTTP/1.1 je nach Anwendungsfall durch die Statuscodes 301 bzw. 307 ersetzt. 302-Weiterleitung ist aufgrund eines Suchmaschinen-Fehlers, dem URL-Hijacking, in Kritik geraten. Webmaster sollten generell von der Verwendung eines solchen Redirects absehen.
303	See Other	Die Antwort auf die durchgeführte Anfrage lässt sich unter der im „Location“-Header-Feld angegebenen Adresse beziehen.
304	Not Modified	Die durchgeführte Anfrage führt zur selben Antwort wie zur vom Client übermittelten Zeit im „If-Modified-Since“-Header-Feld oder sie passt zu dem im „If-None-Match“-Header-Feld gesendeten Entity-Tag. Sie wurde deshalb nicht mitübertragen.
305	Use Proxy	Die angeforderte Ressource ist nur über einen Proxy erreichbar.
306	(reserviert)	306 wird nicht mehr verwendet ist aber reserviert. Es wurde für „Switch Proxy“ verwendet.
307	Temporary Redirect	Die angeforderte Ressource steht vorübergehend unter einer anderen Adresse bereit. Die alte Adresse bleibt jedoch gültig.
400	Bad Request	Die Anfrage-Nachricht war fehlerhaft aufgebaut.
401	Unauthorized	Die Anfrage kann nicht ohne gültige Authentifizierung durchgeführt werden.
402	Payment Required	(reserviert)
403	Forbidden	Die Anfrage wurde mangels Berechtigung des Clients nicht durchgeführt.
404	Not Found	Die angeforderte Ressource wurde nicht gefunden. Dieser Statuscode kann ebenfalls verwendet werden, um eine Anfrage ohne näheren Grund abzuweisen.
405	Method Not Allowed	Die Anfrage darf nur mit anderen HTTP-Methoden (z. B. GET statt POST) gestellt werden.
406	Not Acceptable	Die angeforderte Ressource steht nicht in der gewünschten Form zur Verfügung.

Continued on next page

Tabelle A.1 – continued from previous page

Statuscode	Bezeichnung	Erläuterung
407	Proxy Authentication Required	Analog zum Statuscode 401 ist hier zunächst eine Authentifizierung des Clients gegenüber dem verwendeten Proxy erforderlich.
408	Request Timeout	Innerhalb der vom Server erlaubten Zeitspanne wurde keine Anfrage des Clients empfangen.
409	Conflict	Die Anfrage wurde unter falschen Annahmen gestellt.
410	Gone	Die angeforderte Ressource wird nicht länger bereitgestellt.
411	Length Required	Die Anfrage kann ohne ein „Content-Length“-Header-Feld nicht bearbeitet werden.
412	Precondition Failed	Eine in der Anfrage übertragene Voraussetzung traf nicht zu.
413	Request Entity Too Large	Die gestellte Anfrage war zu groß, um vom Server bearbeitet werden zu können.
414	Request-URI Too Long	Die URI der Anfrage war zu lang. Ursache ist oft eine Endlosschleife aus Redirects.
415	Unsupported Media Type	Der Inhalt der Anfrage wurde mit ungültigem oder nicht erlaubtem Medientyp übermittelt.
416	Requested range not satisfiable	Der angeforderte Teil einer Ressource war ungültig oder steht auf dem Server nicht zur Verfügung.
417	Expectation Failed	Das im „Expect“-Header-Feld geforderte Verhalten des Servers kann nicht erfüllt werden.
421	There are too many connections from your internet address	Verwendet, wenn die Verbindungshöchstzahl überschritten wird
422	Unprocessable Entity	Die Anfrage wurde wegen semantischer Fehler abgelehnt.
423	Locked	Die angeforderte Ressource ist zurzeit gesperrt.
424	Failed Dependency	Die Anfrage konnte nicht durchgeführt werden, weil sie das Gelingen einer vorherigen Anfrage voraussetzt.
425	Unordered Collection	In den Entwürfen von WebDav Advanced Collections definiert, aber nicht im „Web Distributed Authoring and Versioning (WebDAV) Ordered Collections Protocol“.
426	Upgrade Required	Der Client sollte auf Transport Layer Security(TLS/1.0) umschalten.
500	Internal Server Error	Unerwarteter Serverfehler
501	Not Implemented	Die Funktionalität, um die Anfrage zu bearbeiten, wird von diesem Server nicht bereitgestellt.

Continued on next page

Tabelle A.1 – continued from previous page

Statuscode	Bezeichnung	Erläuterung
502	Bad Gateway	Der Server konnte seine Funktion als Gateway oder Proxy nicht erfüllen, weil er seinerseits eine ungültige Antwort erhalten hat.
503	Service Unavailable	Der Server steht, zum Beispiel wegen Überlast oder Wartungsarbeiten, zurzeit nicht zur Verfügung.
504	Gateway Timeout	Der Server konnte seine Funktion als Gateway oder Proxy nicht erfüllen, weil er innerhalb einer festgelegten Zeitspanne keine Antwort von seinerseits benutzten Servern oder Diensten erhalten hat.
505	HTTP Version not supported	Die benutzte HTTP-Version wird vom Server nicht unterstützt oder abgelehnt.
506	Variant Also Negotiates	
507	Insufficient Storage	Die Anfrage konnte nicht bearbeitet werden, weil der Speicherplatz des Servers dazu zurzeit nicht mehr ausreicht.
509	Bandwidth Limit Exceeded	Die Anfrage wurde verworfen, weil sonst die verfügbare Bandbreite überschritten werden würde.
510	Not Extended	Die Anfrage enthält nicht alle Informationen, die die angefragte Server-Extension zwingend erwartet.

Stichwortverzeichnis

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Berlin, 30. September 2012